Ease : A Real-Time Multitasking Executive

David Doyle, B.Sc. Eng.
for the Degree of
Master of Engineering
at
Dublin City University
for
Dr.Barry Mc Mullin, B.Eng., Ph.D.
School of Electronic Engineering
Dublin City University

April 1996

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Engineering is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work

Signed: David M Doyle **ID No.:** 92700888

Date: 15th July 1996

Contents

List of Figures

Abstract

Ease the real time multitasking executive described in this thesis is designed for embedded systems with particular emphasis on DSP motor control applications

Ease provides an application software interface to the underlying hardware and encourages an object oriented programming approach which inherently enhances software integrity, maintainability and dependability in the potentially chaotic real time environment Its focus is to tackle the undesirable aspects of real time programming and device dependent issues thereby allowing the application programmer to concentrate more on the application

The multitasking aspect of the executive means application tasks can be generated with ease which aids development, evolution or enhancement of an application The multitasking aspect also facilitates tasks dedicated to on-line reconfiguration, error handling and fault correction or shutdown procedures

The software quality of a real time application running on the *Ease* platform is paid for by a small percentage of CPU processing power and a larger response time to external events than an unstructured monolithic interrupt driven system

During the course of research, development and prototyping of *Ease* , a number of suitable sample applications have been explored to test and optimise its functionality The most notable of these is the control system for the motor simulation of a shock absorber with an active disturbance load This was implemented as seven concurrent tasks in a uniprocessor DSP system, running *Ease*

Acknowledgements

I wish to express my gratitude to my supervisor Dr Barry Mc Mullin for his professional advice, guidance and encouragement throughout this project

A special acknowledgement must be given to my family for their support and encouragement during the course of my further education

Further thanks are due to my fellow postgrads whom I had the good fortune of working with

Finally I would like to thank Power Electronics Ireland a division of Forbairt whose financial backing made this project possible

Chapter 1

Real Time Concepts

This chapter is a study of the existing body of knowledge through references on real time systems

1.1 What Constitutes Real Time

It can be argued that all practical systems are real time A real time system is characterised by the system responding to occurrences in a dynamic real world environment, within a certain time frame Time is of the essence If the real time system is to be effective then it must respond within given time constraints

In real time systems the integrity of the system's output depends not only on the accuracy of the logical computations carried out but also upon the time the output results are delivered to the external interface This external interface indicates that real time software operates within a highly specialised hardware environment This hardware environment highlights the embedded nature of real time systems The embedded computer system exercises control over a system which is reacting with the real world The computer is essentially within the control loop

A real time application demands from its embedded system not only significant computation and control processing but also, and even more importantly, a guarantee of predictable, reliable and timely operation To be useful the computer system must be deterministic

A definition for real time systems based on the mathematical description

of a system can be found in [9]

> A real time system is a system that must satisfy explicit (bounded) response times or risk severe consequences, including failure

It continues to describe a failed system

> A failed system is a system which can not satisfy one or more of the requirements laid out in the formal system specification

The definition of a failed system means that the system requirements should be known a priori and system operating criteria specified precisely This is particularly true for software as software is often the most volatile and flexible element of a real time computer system Depending on the nature of the critical timing constraints imposed by the external environment real time systems are classified as *hard* or *soft* If failure to meet timing constraints means that the system's performance is degraded but not destroyed then the system is classified as soft If failure to meet timing constraints leads to total system failure then the real time system is classified as hard A *firm* real time system is a system in which a low probability of failing to meet timing constraints can be tolerated

A more formal definition of real time operation is given in [2] quoting [18]

> Real time operation is the operating mode of a computing system, in which programs for the processing of data arriving from the outside are permanently ready in such a way that the processing results become available within a priori given time frames

It should be noted that there is no time information given on the events stimulating responses from real time systems The events triggering responses in the first definition, and the arrival of data in the second definition, may be at randomly distributed instants or predetermined points in time It is the nature of real time environments to be potentially chaotic Taming and controlling these environments through on going interaction over time is the goal of real time systems To achieve this, prediction, measurement and reduction of event response times is paramount

1.2 Hardware Issues

Embedded real time systems have by definition very specialised hardware platforms The most notable aspect of this specialisation is the interface devices which interact with sensors, the controlled process, user interface and possibly an overall integrated computer system governing many embedded systems

To successfully implement a practical real time system requires a sound understanding of the underlying hardware Issues such as CPU performance, interrupt facilities, memory space, language support and development tools all have to be considered

Embedded platforms tend to make efficient use of hardware resulting in minimal standalone hardware systems There is a minimal (or non existent) amount of peripheral devices such as disk drives and printers Embedded platforms are dedicated, therefore human interface is generally handled by an external computer system Embedded systems are autonomous and therefore must contain substantial code and system parameters in read only memory

Reliability is also a very important issue with real time hardware The hardware must be fault tolerant The need for reliability sometimes necessitates the need for employing identical redundant systems which can continue operation if one or more of the systems fail [19] [20]

1.2.1 Real Time CPU

The choice of microprocessor for an embedded systems is important both from a performance and functional point of view

The following are some real time concerns for the CPU

- Performance Processing Power of the Microprocessor
- Instruction Set A rich instruction set allows compilers to make efficient use of code, for example floating point provision
- Language Support High level/Real time language support
- Interrupt Mechanisms Support for interrupt events

- On Chip Hardware Facilities Some processors have on chip timers and memory

The real time CPU must have the appropriate processing power to perform the necessary computations within the system's real time timing constraints Efficient coding of real time programs greatly enhances the ability of a system to satisfy timing constraints Efficient code makes the best use of the available CPU power The ability to generate efficient code is greatly enhanced by the richness of the instruction set and addressing modes

There are two different schools of thought on processor instruction sets In a *complex instruction set computer* or CISC there are many instructions some of which may be implemented by microcode within the micro processors hardware In this way complex functions are performed in hardware and memory use is reduced The decode and execution time for every instruction increases however for CISC The advocates of *reduced instruction set computers* or RISC machines argue that by simplifying the instruction set, instruction execution speed can be increased Most compilers generate code with heavy use of a small number of instructions such as LOAD, STORE, ADD, SUB and branch instructions RISC machines eliminate the CISC disparity that the execution times for all instructions are increased for the benefit of a few seldom used instructions RISC machines rely on the compiler to generate efficient code whereas CISC relies on the speed of hardware microcode to compensate for increased instruction times In real time systems RISC machines have the advantage that the longest and average instruction execution time is reduced A discussion on the application of RISC processors to real time systems is given in [12] while [13] presents a more theoretical treatment

1.2.2 Memory

Memory issues have an impact on all measures of systems performance The most important memory issue for real time systems is access time Memory access that is slower than a CPU clock cycle forces the CPU to wait a number of clock cycles to access the code or data This significantly increases instruction fetch and/or data load and store operations Time critical data

structures and code should reside in fast memory Real time programs should utilise internal registers and immediate addressing modes where possible to minimise memory access Embedded computer systems must also contain volatile and non volatile memory to be autonomous

1.3 Language Issues

The following section is an exploration of the programming language features that are desirable in real time applications Some of the features are desirable not only in real time systems, but in all well structured, reliable, maintainable and efficient software systems Real time software systems are a special high performance subset of all software systems where software quality is extremely important

1.3.1 Modularity

A language that facilitates modular programming is highly desirable in real time systems Modularity promotes data encapsulation[7] If each module has its own local data and has a well defined interface then there is less chance of unpredictable data corruption by other functions There is also the benefits of applying a structured approach to software analysis, design, coding and unit testing The internal workings of the module are invisible to the function calling it This aids maintainability as modifications can be localised to particular modules All these factors are highly desirable for real time systems

Parameter passing between modules can be achieved by several methods including the use of global variables[1], call by value or call by reference

Parameter passing through call by value or call by reference typically involves the parameters or pointers being passed on to the stack which can have a significant execution time impact for real time programs Global variables do not have the same execution time penalty but do have an impact on software quality Parameter passing can sometimes increase interrupt latency as many compilers disable interrupts during parameter passing

[1]Global variables are external parameters directly accessible to a number of modules

The call by value mechanism copies the actual value of the data to the called function It works well when there is a test being performed on the data or when the data is the input to a mathematical function The mechanism is designed to ensure that the input data is not changed by calling the function

The input data of the call by reference mechanism is a pointer to the data which the called function must access This mechanism is designed to allow the input data to be changed by the called function Each access to the input data by the called function requires at least one level of data indirection which has a performance impact When the input data structures are sufficiently large the call by reference mechanism has the advantage over call by value in that only a pointer to the input data structure needs to be placed on the stack rather than the entire structure as in call by reference

1.3.2 Recursion

Recursion is a mechanism provided by many programming languages whereby a function can call itself This mechanism allows the programmer to write elegant and concise code but in general has an adverse effect on real time performance

The execution time for allocation and de-allocation of parameters and local variables is costly to real time programs which should be as efficient as possible to meet timing constraints The use of recursive functions makes the run time memory requirements very difficult to analyse

1.3.3 Re-entrant Procedures

A re-entrant function is one which can be called by a number of concurrently running tasks Functions of this type are necessary if concurrent tasks need to share the same code Re-entrant functions may not use any data that is in a fixed location but must use memory which is dynamically allocated for each call This allocation is either on the stack or through a memory allocation scheme such as the `malloc()` procedure in C[11]

Awkward schemes must be employed if code is non re-entrant and needs to be shared by two or more concurrent tasks For example if two or more tasks need to use a particular non re-entrant function they may avoid the

problem by each having an exclusive copy of the same function The problem with non re-entrant functions is that they contain data at a fixed location which may be overwritten each time the call is invoked

1.3.4 Data Typing

Strongly typed languages force the application programmer to be precise about the way data is handled which is beneficial to real time programs Typed languages require that each variable and constant be declared as being of a specific type and that this declaration is made at compile time

1.3.4.1 Abstract Data Typing

It is important be able to represent abstract ideas concisely in computer languages as well as in human ones Languages that allow abstract representation of entities which comprise of different data types makes program design easier and aids program comprehension When moving from the analysis and design stage to coding, effective mapping of information models is made much easier by provision of abstract data typing Finally parameter passing to functions is made clearer An example of an abstract data type is the `struct` specifier in the C programming language[11] C also allows user defined types through the `typedef` declaration

Real time performance may be degraded by using abstract data types, however its benefits to software quality and maintainability are significant For example the C `struct` specifier is used to logically group data elements Accessing a data element in a structure requires knowledge of the address of the structure and the displacement of its element If the data element was not in a structure then only the address of that element would need to be known and there would not need to be a displacement calculation

1.3.5 Assembly Languages

Assembly languages lack most of the desirable features of high level languages Assembly languages are tedious, unstructured and vary for different machines They do have the advantage however of possessing a more di-

rect control of hardware and possibly being more efficient than a high level equivalent

Assembler programming should therefore be limited to use in extremely time critical applications or for controlling hardware features not supported by the compiler In real time systems assembly programming is often necessary to access elements of the highly specialised embedded hardware environment In general assembly language should be avoided where possible but typically embedded real time systems will contain some inevitable mix of high level and assembly language

1.3.6 Object Oriented languages

Languages which are designed to encourage a high degree of data abstraction and information hiding are called *object oriented* languages Object oriented programming techniques show significant advantages in improving overall system quality at all stages from mapping the problem domain to a robust, quality specification, through design and over the life of a particular software application[7][8]

Object oriented languages provide many features necessary to encourage good software engineering technique Function *polymorphism* for example allows the programmer to create a single function which operates on different objects depending on the object involved Object *inheritance* allows the programmer to create new objects in terms of existing objects

The increase in software quality is paid for by a significant time penalty which may be too severe for many real time systems For example in one study, code written in objective C, (an object oriented variant of C) was found to be 43% slower than the same application written in conventional C or programs written in the object oriented language Smalltalk are known to be approximately 5 to 10 times slower than those written in conventional C[9]

In many cases, particularly in real time systems where software quality is paramount the benefits of object oriented techniques may make it worth employing more powerful processors to overcome the time penalty associated with object oriented systems

1.4 Strategies for Real Time Scheduling

An operating system is a collection of specialised system programs The *Kernel* or *Nucleus* holds the minimum functionality required for an operating system The Kernel must perform three core services task scheduling, task dispatching and inter-task communication The scheduler module determines which task is to be run at a given time in the system The dispatcher does the necessary bookkeeping to activate the next task to be run and stores the context of the last task The inter-task communication module handles data interchange and synchronisation between the tasks within a system

1.4.1 Busy Waiting

A busy waiting scheme is not strictly speaking a real time kernel but it warrants mention as it is the simplest way in which a real time computer system can respond to an event It comprises simply of a repetitive test to establish whether or not an event has occurred If the event has occurred then a process is invoked to deal with it, if it has not then the same test is repeated

Busy waiting or *polled loop* systems have a number of features desirable for real time systems namely

1 They allow for very fast response times (for single devices)

2 They are easy to write and debug

3 The response time is easy to calculate making the system event deterministic

However polled loop systems also have a number of general disadvantages which are unacceptable in many applications They are as follows

1 They require the processor to be dedicated to monitoring a single event

2 They can't operate in a multitasking environment as only a single task is allowed

3 The polled loop is a waste of CPU processing power

1.4.2 Coroutines

Coroutine or *co-operative multitasking systems* are used widely in soft real time systems[2] A coroutine scheme allows applications to be written in a multitasking environment but requires disciplined programming and an appropriate application Coroutines are implemented by breaking processes up into discrete code segments or phases The phases are organised in a way that allows each process to be temporarily suspended before completion without the loss of critical data At the end of each phase there is a call to a central dispatcher which decides which process to run and keeps a record of which phase the process is at Any data that needs to be preserved between dispatches must be stored in global variables Communication between processes is also via global variables

If the process phases have a known execution time then response times can be determined Another point of note is that coroutines operate without hardware interrupts The main disadvantages with coroutines are that not all processes can be easily broken down into phases, communication via global variables is not desirable and finally coroutines places great demands on the programmer to use the scheme correctly

1.4.3 Interrupt Driven Systems

In interrupt driven systems, scheduling is achieved through hardware or software interrupts The interrupt informs the real time system of the occurrence of an event Dispatching the appropriate task is conducted by the interrupt service routine in a single interrupt system or directly via hardware in a multiple interrupt scheme There must be an idling program for the system to revert to when all events have been serviced

The events which drive the tasks of an interrupt driven system can be sporadic, periodic or some combination of both Systems in which only periodic events occur are called fixed rate systems Systems which must respond to both sporadic and periodic events are called hybrid systems

[2]For example, in Microsoft Windows programming

1.4.3.1 Context switching

The most fundamental mechanism in interrupt driven systems is context switching Concurrency is achieved through a principle called pseudo-parallelism[4] A number of separate concurrent tasks can be run on the same processor if all the information relating to the state of any particular task can be stored so the task can be resumed after being interrupted The context is essentially the image of a virtual processor on which each task exists The real time kernel makes the context switching invisible to the application task

The price of concurrency is the context switching overhead which is a major contributor to event response times Context switching times must be minimised because any cycle wasted in the kernel is of double loss to the application as no useful work is being done As a general rule only the minimum amount of information required to safely restore task context is saved

The following information is generally what is saved as part of a context switch

- The contents of the program counter
- The contents of the processor's registers
- The contents of coprocessors registers (if any)
- The contents of memory page registers
- Memory mapped I/O location mirror images
- Special variables

Normally interrupts are disabled during the critical context-switching period Sometimes however after sufficient context has been saved, interrupts may be re-enabled after a partial context switch in order to handle a burst of interrupts, to detect spurious interrupts, or to handle a time overloaded condition

1.4.3.2 Context switching using the stack

A typical method for saving and restoring context in a multi tasking system is by using the stack The TMS320C30 compiler for example generates code to push all the processor registers used by an interrupt function on to the stack upon entry to the function Conversely it pops these registers off the stack to restore the original context upon function exit

The stack is a resource with a limited memory allocation The amount of stack space that a program consumes and releases swells and recedes at run time The storage of task context also places an extra strain on the stack There is a danger of the stack memory being exhausted by the combined stack usage of a number of tasks particularly if interrupts are not disabled during interrupt routines If interrupts are not disabled during interrupt routines then a number of interrupts occurring or even a burst of the same interrupt will cause a number of contexts to be stored and may lead to stack overrun

The dynamic nature of the stack usage makes it very difficult to determine how much stack space a program will consume before an event and how much stack space the event handler will consume after the event The real time system must take into account the combined worst case stack usage in order to maintain system integrity at all times

1.4.3.3 Round Robin Systems

The *Round Robin Scheme* is characterised by having several tasks which are executed sequentially to completion often in conjunction with a cyclical executive The fundamental idea behind round robin is that each task is assigned a fixed quantum of processor time called a *time slice* A fixed rate clock generates an interrupt corresponding to the end of a time slice If the task does not complete within its time slice then the context is switched and the task is placed at the end of an executable list It is assumed that all tasks have an equal priority The round robin scheme is the fairest way to allocate processor power between tasks of equal priority It must be noted however that round robin systems do not respond to external interrupts but only the system clock interrupt Real time response times are hard to calculate

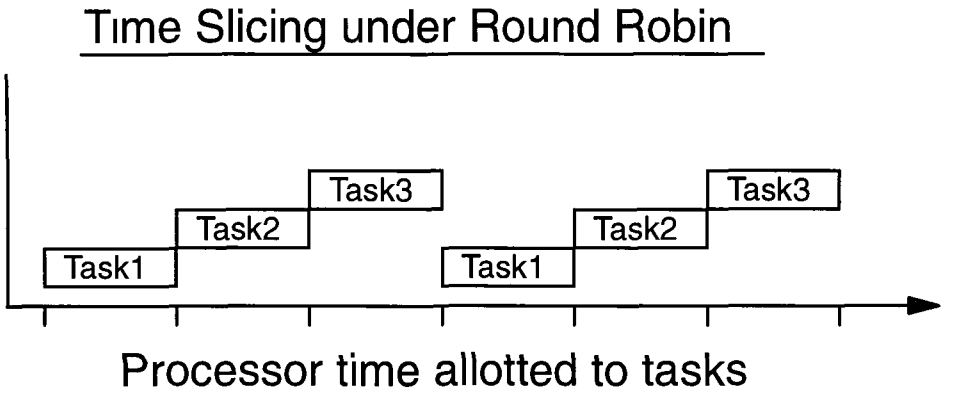


Figure 1 1 Time slicing under Round Robin

as they are a function of the length of the time slice and of the number of executable processes on the executable list

1.4.3.4 Preemptive Priority Systems

In preemptive priority systems scheduling is achieved through assigning real time events with a particular priority This allows the tasks which need processor attention to meet their deadline to interrupt tasks of lower priority The priorities of these tasks may be *fixed* or *dynamic*

The scheduling priority of a task may not necessarily reflect how critical the task is to the system For example in rate monotonic systems priority is assigned based on the execution frequency of tasks A task driven by an event with a short period is assigned a high priority, however this task may not be the most important task to the system This is a phenomenon known as *priority inversion* This distinction is of no concern when all the tasks must meet their deadlines, however in many real time systems transient overloads may occur and it may not be possible to meet all deadlines When such an overload occurs then it is vital that critical tasks meet their deadline even at the expense of less critical deadlines In this way system integrity has a better chance of being upheld even after transient overloads When a lower priority task is denied resources in this way through a higher priority task preempting, the lower priority task is said to be facing a problem known as

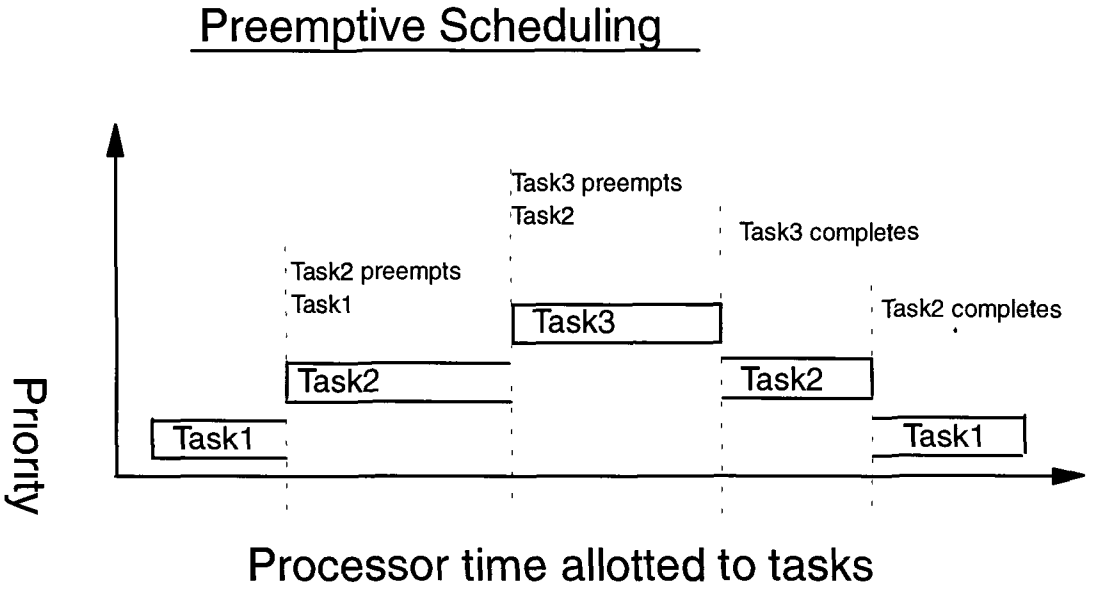


Figure 1 2 Preemptive Scheduling

starvation

1.4.3.5 Hybrid Interrupt Systems

There are many versions of interrupt only systems Hybrid systems incorporate both the fixed rate and the sporadic interrupts which typically are present in an embedded application

A special type of hybrid system uses a combination of round robin and preemptive systems In such a system tasks of equal priority may run concurrently in round robin fashion while a higher priority task can preempt a lower priority one

Interrupt only systems have the advantages that they are easy to write and code efficient They have typically very fast response times as scheduling is achieved via hardware

The disadvantages of interrupt only systems is the processing power wasted in the idling task and the difficulty in providing advanced services These advanced services include interfaces to devices and multi-layered networks Another weakness is the system's vulnerability to timing variations

and unanticipated race conditions

1.4.4 Foreground Background Systems

The foreground background model of real time scheduling systems sums up all the systems discussed so far The foreground of the model comprises a number of interrupt driven tasks The background is used by non time-critical tasks The background tasks can always be preempted by any foreground task The background processing power can be used to perform low priority self testing or performance testing

1.4.5 Task Control Block Model

The task control block model is a technique for representing and controlling a multi tasking system It is quite popular in commercial, full featured, real time operating systems It has the advantage that it can cater for a variable number of tasks and that tasks can be created dynamically The main disadvantage of the task control block model is that when the number of tasks created is large then the kernel overhead becomes significant making the system unwieldy

The task control block technique hinges on assigning each task with an identification string or number, a status, a priority and space to store the task's context These items are stored in a structure called the task control block Each task control block would typically be grouped in a larger data structure such as a linked list

1.4.5.1 Task Management

The operating system manages tasks in the system on the basis of the information stored in the task control block The task control block is updated upon any scheduling event In a uniprocessor system there can be only one task executing at any one time There are three different states that the status field of the task control block can be

They are

- *running*

- *ready*
- *blocked*

The *running* task is the one which is currently allocated the CPU A *ready* task is one temporarily blocked to let another task run A task's status would be set to ready if it was preempted or if its time slice had expired

Blocked tasks are ones which are not selected as ready Tasks are made ready by the operating system upon a certain event An event is either a hardware interrupt or a software trap Certain systems also have a *dormant* task state The dormant state is used by systems in which the number of tasks is fixed and where the task control block is allocated for all possible tasks A dormant task is one which is not yet created or available to the operating system

Every event or system level call is made via the operating system The operating system decides the next eligible task to run after a scheduling event, releases the CPU from tasks when their time slice has expired, arbitrates on the allocation of resources and facilitates inter task communication and synchronisation

1.5 Inter Task Communication and Synchronisation

The integrity of data transfer between tasks and the synchronisation of tasks both internally and externally poses a problem in any multitasking system The system must guarantee not only that data is transferred correctly but also that certain sequences of events must never occur

1.5.1 Data Sharing

The simplest and fastest way to pass data between tasks is via shared memory Compilers can generate very efficient code for accessing data in this way as it only requires knowledge of an address in memory The disadvantages of using shared memory between a number of tasks is that the shared data is prone to corruption This can occur for example through a task operating

with shared data which is preempted by a higher priority task which updates the same data The preemption could occur while the former task is mid way through a calculation using the data, yielding unpredictable results The use of shared memory for intertask communication requires the system to have some other synchronisation mechanism such as semaphores built into the code to guarantee system integrity

1.5.1.1 Double Buffering

Double buffering is used when time relative data needs to be transferred and the producer generates data at a slower rate than the consumer processes it It is commonly used in systems such as telemetry The basic idea is that there is shared memory divided into two blocks At any time there will be one block updated by the producer and the other can be accessed by the consumer A hardware or software switch is used to alternate between the two buffers Double buffering is also commonly known as *ping pong* buffering The consumer must consume data faster than it is produced for this system to work

1.5.1.2 Ring Buffers

Ring buffers are an extension of the above double buffering scheme where there is more than two buffers for the producer to fill The system operates on a FIFO queue scheme The FIFO system allows the consumer to have more time before servicing the queue This system is commonly used in a system such as a data logger where the time afforded by the FIFO depth allows the system to write to disk

1.5.2 Message Passing

Message passing is a scheme in which tasks can transfer data via the operating system through calls to send and receive The data is transferred to a mutually agreed upon memory location which is generally cleared after the operation Synchronisation is achieved through task rendezvous A sending task is blocked until there is a receiver present to take its message Con-

versely a receiving task is blocked until there is a sending task present to generate a message

If there are multiple readers and writers then the identities of the blocked tasks are recorded by the operating system The operating system chooses which of the blocked tasks will rendezvous if there are a number of tasks blocked pending on a rendezvous partner Null messages may be passed for pure synchronisation purposes

Some executives may convert external events into messages which the tasks may synchronise with

1.5.3 Semaphores

Critical regions are identified as being the sections of code in tasks which access resources which can only be used by one task at a time These resources include shared memory, certain peripherals and the CPU itself One of the main thrusts of task synchronisation is ensuring that certain sequences of events don't occur such as two tasks entering their critical regions and accessing the same resource Dijstra[1] put forward a scheme for protecting critical regions in multitiasking systems which make use of a special variable called a *semaphore*

The semaphore is basically an unsigned counter and there are two operations which can be performed on it *up* and *down* An up action on a semaphore will increment the value of that semaphore A down operation will decrement the value of a semaphore or block the process which made the call if the semaphore is zero

The up and down operations on semaphores are atomic which means that no other process can access the semaphore until the semaphore operation is complete This is essential to avoid race conditions and solve synchronisation problems

A process will never be blocked if it does an up operation If there are any tasks blocked on an unsuccessful down operation the up operation will free one of these tasks

Semaphores are a very versatile synchronisation mechanism They do require an effort on the application programmer's part to identify critical

regions, choose and maintain appropriate semaphores and embed semaphore operations in the application's code

1.6 Conclusion

A real time system interacts with, or reacts to, a dynamic real world environment The system's integrity depends not only upon the systems logical correctness but also upon a timely response to external events Real time systems must be predictable, reliable and timely to be useful Precise system specification is especially important for real time systems

Real time software systems operate in specialised hardware environments These hardware environments would typically have connections to sensors and actuators to interact with or monitor a real process It is important that the application programmer has a sound understanding of the underlying hardware as CPU performance, interrupt facilities and memory issues all have a direct impact on real time performance

Real time software systems are a high performance subset of all software systems The software for real time systems should be well structured, reliable, deterministic, maintainable and efficient The concurrent nature of real time systems means that there is some degree of multitasking in the software Software quality is particularly important for real time systems

If there are a number of concurrent processes in a real time system the software system will benefit from a real time operating system The real time operating system facilitates the structuring of the processes and activities of an application into dedicated tasks This has the advantage of making the solution to the application modular The second advantage of a real time executive is that it structures interactions between application tasks by handling intertask communication and synchronisation in a safe manner (minimising the chances of unanticipated race conditions) The third advantage of an executive is that it provides an application software interface to the underlying hardware This API provides an event interface mechanism for tasks

Chapter 2

Executive Features

2.1 Desirable Features

The computer in a real time embedded system is essentially within the control loop and its responsibilities in that role are its primary functions Synchronisation, scheduling and communication between the different components of real-time software in a reliable, timely and predictable fashion places great demands on the software The real time environment requires a number of features from the software, many of which fall in the domain of the executive

2.1.1 Fundamental Features

There are three fundamental features which reflect what is essential in real-time systems[5][9][2][10] They can be itemised as follows

- Concurrent Processing
- Hardware/Event Interface
- Interprocess Communication and Synchronisation

2 1.1.1 Concurrent Processing

All real time systems must facilitate concurrent processing The level of concurrent processing is a function of the number of external events that must be handled by the system and the natural parallelism of processing

within it The executive must ensure that the multi-tasking scheme is flexible and efficient It should be deterministic for critical tasks

2.1.1.2 Hardware/Event Interface

All real time embedded systems must have a significant interface to the underlying hardware The real time embedded system may be defined as being in permanent contact with an active environment If the executive is designed in a modular fashion then very specific hardware interface modules could be added, modified or removed without affecting the core of the executive The real time system must be able to respond to events and the executive must support an event interface to the tasks that make up its application

2.1.1.3 Interprocess communication and Synchronisation

The real time system must facilitate inter task communication and synchronisation It must do this in a way that avoids data corruption and race conditions to maintain software quality

2.1.2 Quality Attributes

The quality attributes tend to influence the way in which systems should be developed They also influence the design but are independent of the required functionality[5]

- Dependability
- Reconfigurability
- Usability
- Certifyability
- Constraints
- Evolution Capability

2.1.2.1 Dependability

Dependability is a measure of how much reliance can be place on the quality of service that a system can deliver It is a function of reliability and maintainability

2.1.2.2 Reconfigurability

This is a property of a system which expresses the possibility of being able to influence the structure and/or functions during system operation If a system is capable of changing its properties without degradation of its services then it can be qualified as reconfigurable on-line If modifications of its properties necessitates temporary interruption of its services then the system is reconfigurable offline

2.1.2.3 Usability

This system feature is related to ease of use of a system by its end user

2.1.2.4 Certifyability

Certifiability of a system expresses the possibility of obtaining a formal statement of compliance of system operation with respect to its specified requirements

2.1.2.5 Constraints

This is a system property which measures the ability of a system to comply with non-functional or physical constraints These may be characteristics such as size, power consumption, price, colour and temperature range for example

2.1.2.6 Evolution Capability

This is a measure of how much a system is designed to evolve over its life cycle

2.2 System Specification

The thesis so far has been a study of what is essential, desirable and reasonable in real time systems However the application domain ultimately defines the specification of the executive The target application considered in this thesis is that of DSP servo motor control The ultimate application to which the executive provided a software platform was a research servo motor test bed This application required the following features

- Control frequencies of up to 4 kHz
- Real time user interface and display
- On line reconfigurability of system parameters
- Control of an active load
- Intensive mathematical computation in control algorithm

It is clear from above that the executive must facilitate multiple concurrent application tasks These tasks must interact with other and the external environment The executive must be structured in a way that guarantees response deadlines for time critical tasks in the system The core features to be provided are as follows

- Preemptive event driven scheduling
- Synchronisation and communication facilities for application tasks
- Handling of device interrupts

Each task in the application should be a separate programming entity with its own exclusive data code and stack The executive must provide for communication between these exclusive memory areas

The executive must also provide a means of synchronisation internally between tasks and externally with the real time environment To guarantee response deadlines of time critical tasks requires a priority scheme where critical tasks can preempt tasks of lower priority The scheduler is therefore

activated on a scheduling event which may be internal or external Finally the executive overhead must not be too great for the application and the hardware platform

2.3 System Design

The executive considered in this thesis is essentially a software platform facilitating concurrent application tasks and providing application services The executive is designed for an embedded hardware platform, therefore the main thrust of the design is to make it a minimal kernel This design strives to make the kernel fast and efficient in order to reduce overhead and to meet timing constraints Simplicity is chosen as a fundamental design principle as it inherently makes the executive more predictable, dependable and optimal by not allowing unwieldy complexity to creep in It is important that the internal workings of the executive are understood by the application programmer The executive was christened *Ease*

Unlike commercial operating systems the embedded environment typically does not have to handle device drivers for devices such as disk drives and complex user interface devices The embedded system can however have many varied and application specific device interfaces For this reason a design decision was made that hardware interface would be carried out directly by the application tasks and not via system calls *Ease* still provides an event interface for application tasks This approach makes *Ease* smaller, less complex and provides more flexibility for the application programmer

An embedded system typically would have a known number of tasks and these would each be assigned a priority Static process priority was chosen over dynamic process priority as dynamic process priority may obscure application bugs For the embedded system there is also no need for dynamic process creation and destruction

To make *Ease* as platform independent as possible it was written in C where possible and assembly where necessary All executive components were coded in a modular fashion to aid evolution, development, addition or enhancement of its services All executive services are accessed through C callable functions

The message passing scheme was adopted for inter-task communication and synchronisation Message passing inherently incorporates communication and synchronisation in one mechanism It was decided that external events should be converted to messages by *Ease*

The design goals may be summed up as follows

- Optimise speed and efficiency but not at the expense of design comprehension
- Keep executive functions small and fast
- Allow all program components to co-operate with each other with minimum overhead
- Choose to spend memory to gain speed

2.3.1 Scheduling Scheme

The fundamental uniprocessor method for introducing concurrency involves pseudo-parallelism This is achieved by the executive switching processor context between independent task objects These tasks have one of three states ready, running or blocked Tasks are not created dynamically The executive keeps track of task states in a task control structure

Task scheduling under *Ease* is conducted on a priority basis with a time slicing scheme for tasks of equal priority The scheduler is run upon a scheduling event which is an event which changes the state of an application task The scheduler is triggered by one of three conditions The scheduling event may originate from an external source such as an interrupt The scheduling event may originate from the *Ease* system clock which indicates that a task has expended its time slice Finally the scheduler may be invoked from a task seeking to send or receive a message The call to send/receive will result in an application task changing state either through rendezvous, which will make another task ready or by blocking the task making the call if there is no rendezvous partner Scheduling is guaranteed at a minimal level by the *Ease* system clock The actual scheduling mechanism is designed to be as fair as possible without excessive overhead

The scheduling mechanism does the following on a scheduling event

- Make a limited context switch so the scheduler can run
- Run scheduler to decide which ready task to select on the basis of the relevant states of tasks within the application
- Update the task control structure on the basis of the scheduler's decision
- If the same task is to be run restore it
- If another task is to be run do a full context switch

2.3.1.1 Task Control Structure

Ease keeps track of task states through a task control structure An element corresponding to a single application task of this task control structure contains the following information

- Blocked status running ready or blocked
- Quantum tick the amount of time slices that the task has run for
- Stack pointer pointer to task's exclusive stack
- Task Id An integer to identify application task
- Root Function pointer to root function of the task
- Next member pointer to next task of same priority

Ease maintains closed linked lists of tasks of equal priority *Ease* keeps an array of pointers to the current task at each priority level A scheduling event which may cause a task of any priority level to be unblocked causes the *Ease* scheduler to scan through all of these linked lists starting at the list corresponding to the highest priority tasks The time slicing clock will cause a scheduling event which will not change the current priority level In this case *Ease* will advance the quantum tick of the task and check to see if it has expended its share of CPU time If the task has expended its quota of quantum ticks then the executive will run the next ready task on the list of tasks at that priority level

2.3.2 Inter-task Communication and Synchronisation

As message passing inherently incorporates communication and synchronisation in the same mechanism *Ease* employs message passing for inter-task communication and synchronisation This approach slightly penalises tasks that only want a synchronisation service but has the advantage of making the mechanism generic without the need for another service for communication If tasks require only synchronisation then a null message is passed

The *Ease* message passing scheme is designed to transfer data between the application tasks' exclusive memory areas This service may be ignored in lieu of another method such as the use of global variables but this forsakes the advantages that message passing gives The message passing scheme copies the message data from the sender task's exclusive memory area to the receiver's Message passing ensures that there is no data corruption and structures the application in a way that improves software quality and maintainability

Ease also links event handling into the message passing scheme by converting external events into messages which application tasks can respond to This approach makes the mechanism of interface to external events invisible to application tasks It also makes synchronisation to internal and external events generic to application tasks

Ease keeps track of tasks sending and receiving messages through a communication *channel* structure These communication channel structures are basically queues of tasks which are seeking rendezvous partners The queue can be either empty, be a queue of senders or a queue of receivers A task that calls send/receive will be put on the channel queue until a rendezvous partner makes the converse call Application tasks nominate which channel they wish to conduct message passing over in their calls to send and receive Tasks which seek a rendezvous partner are in a blocked state

From the above it is clear that an unbuffered message passing scheme was chosen over a buffered scheme The reason for this design decision is firstly that the buffered scheme makes the executive unwieldy and secondly that it does not make sense to buffer most if not all real time events that an application task may wish to respond to For example there is no sense

in buffering interrupts that must be serviced within a specified time if the interrupts are not serviced each time, that indicates a system failure Adding buffering would increase the executive's complexity and overhead for only minimal extra advantage

Another consideration regarding message passing is whether the message data has fixed or variable size The fixed sized message buffers have the advantage of not needing size information in message calls Variable sized message buffers have the overhead of size checking to ensure that the sender's message is not too large for the receiver The fixed size message scheme makes the entire real time system more predictable as each message package takes a fixed length of time to transfer The variable size message passing scheme increases the maximum interrupt latency as the amount of time to transfer a message varies and interrupts are disabled during the critical sections of message passing The worst case interrupt latency in the variable sized scheme is the time taken for the largest message to be passed The variable sized scheme has the advantage of decreasing average interrupt latency if the message size varies over a range of sizes Fixed size message passing has less overhead, more predictability and less flexibility than variable sized message passing The variable size message passing scheme was chosen for *Ease* for the flexibility aspect of it

All tasks at all priority levels have access to communication channels All external events can each be attached to a specified communication channel When the event occurs *Ease* will send a predefined message on that channel The *Ease* communication channels support multiple senders and receivers *Ease* does not guarantee which task on the channel's queue will rendezvous first if there is a number of tasks on that queue

2.3.2.1 Communication Channel Information Structure

Ease uses the communication channel information structure to keep track of message passing in the system The structure is simply a queue of three elements task identification, a pointer to memory and a size The three elements have different meanings depending on whether the channel is currently maintaining a queue of senders or receivers

If the channel has a queue of senders then the task identification is a unique integer to identify the task which is blocked on send The pointer to memory is a pointer to the task's private memory where it has prepared a message The size element is the size in 32-bit words of the message that it wishes to send on that channel

If the channel has a queue of receivers then the task identification is a unique integer to identify the task blocked on receive The memory pointer is a pointer to the receiver's private memory area where it wishes the incoming message to be placed The size element is the maximum size of message in 32-bit words that the receiver wishes to take

In the case of rendezvous *Ease* will know which task to unblock *Ease* will also have to make a decision based on the sender's and receiver's size arguments There can be two occurrences which would result in a message not being passed The first is if a sender is queued and the receiver it is about to rendezvous with has a maximum message size smaller than the sender's message size The second is if a receiver is queued and the sender's message size is greater than the receiver's maximum message size Both conditions will result in the call to send or receive failing and *Ease* indicating that the sender's message size is too large for the receiver Passing a message of incompatible size generally indicates an application error However the system may recover from the situation gracefully as follows If the receiver is queued the sending task can reduce the size of its message until the call succeeds If the sender is queued then the receiver could increase its buffer size until the sender's message fits Alternatively one could just let the call fail until another task with the appropriate size seeks to rendezvous

2.3.2 2 The Send Call

The send call is the application task's interface to the *Ease* message passing mechanism The task nominates the channel it wishes to send the message over with this call It also passes a pointer to the memory where the message resides and the size of the message The call returns the standard success or failure code depending on the success of the call A failure can be an invalid channel or that the message size is too great for the receiver The call

also returns a pointer to the rendezvous task's root function to identify the rendezvous partner The root function will be discussed in section 2 3 3 1

2.3 2 3 The Receive Call

The receive call is the converse call to send The receiving task nominates the channel it wishes to receive over with this call It passes a message pointer and maximum message size to *Ease* via this call Receive returns whether the operation was a success the actual size of the message received and a pointer to identify the rendezvous task The call will fail if an invalid channel argument is passed to it or if the sender's message size is too great

As event generators are virtual tasks *Ease* returns a NULL task pointer if the sender is an event converted to a message by *Ease*

2.3.3 Initialisation

Ease requires a number of application specific details which it must have access to upon system initialisation For example, the number of tasks the executive must deal with is application specific *Ease* needs access to information provided by the application programmer on the number of tasks, their priorities, their stack allocations and a way to uniquely identify each task This information will be placed in the executive's task control structure There may also be a number of other hardware specific initialisation procedures such as the initialisation of the system clock for time slicing In essence *Ease* requires both private and application specific initialisation

To provide application specific setup in a flexible way, *Ease* requires the application programmer to write an application specific C function which generates tasks and sets system timers This approach also provides a framework for adding of any future initialisation processes which may be needed The implied target platform specific initialisation is handled in this function also

2.3.3 1 Task Creation

The application specific initialisation function will consist of calls to create tasks The application programmer passes three arguments in the create task

call These are the name of the root function of the task, its priority and the stack space the task must be allocated The root function of a task is the C function which represents that task The root function typically consists of an endless loop

The application programmer must be careful in choosing the stack allocation If the stack allocated is too small then the task stack may overrun and corrupt data, on the other hand if the sum of the stack allocations are too great then there may not be enough physical memory The stack must be able to accommodate all the local variables of the task's root function plus those of any functions which are subsequently called to the deepest nested level and must cater for memory taken by the actual parameters passed on the stack

2.3.3.2 Timer Initialisation

In this same initialisation function a call to set all timers used by the application must be called The application programmer passes one argument in the timer initialisation call This argument is simply the frequency

A particular timer on the target system has the special function of generating time slice ticks for application tasks and must be called regardless of whether application tasks wish to avail of system timer services or not The period time slicing tick must also be set to an appropriate value which is generally recommended to be ten times the time it takes to switch context

2.3.4 Services

The executive services are made through C calls The most fundamental calls are send and receive, discussed above The other calls are calls to set timers and cause events to generate messages

2.3 4.1 Timer Services

All timer service functions are called directly by the application tasks They consist of three arguments

- Ticks

- Channel
- Mode

With the channel argument, the task nominates the channel on which a message will be sent when the timer has expired

The ticks argument is the number of timer events that the executive must wait before sending the message The tick period is set by a timer initialisation call in the application specific initialisation routine

The mode argument informs *Ease* of the mode of the timer The mode may be monostable or astable

2.3.4.2 Event Services

The initialisation functions of event services are simpler than the timer services in that they have only one argument This argument is simply the channel which the executive will send a message over when the event occurs Each event to which *Ease* provides event services has it's own initialisation function defined in `ease h`

2.3 4.3 Error Services

Ease indicates errors through leaving an error message string at a specific global location in memory This location can also be accessed by application tasks An application task detecting a serious error can copy its error message string to this global location

The executive also gives application tasks access to special variables through the include file which gives information on interrupts which are lost through no task being ready to respond to them

2.4 Conclusion

Ease , the executive of this project, is specified for DSP motor control applications This application domain demands certain features and quality attributes from the executive *Ease* is specified to support multiple concurrent application tasks and facilitate preemptive event driven scheduling

Ease is designed to be small, fast and efficient as it is to be used in an embedded environment

Task scheduling under *Ease* is conducted on a priority basis with a time slicing scheme for tasks of equal priority *Ease* employs message passing as a means of intertask communication and task synchronisation as message passing incorporates both in the same mechanism *Ease* also links event handling into the message passing scheme by converting external events into messages which application tasks can respond to This approach makes the mechanism of interface to external events invisible to application tasks It also makes synchronisation to internal and external events generic to application tasks

Chapter 3

Ease Software Design

As the application domain of *Ease* is the embedded environment *Ease* is designed to be fast and efficient to reduce overhead and meet the timing constraints imposed on it It is designed to be compact so as not to take up too much memory, as memory is quite a scarce resource in embedded systems These optimisations are not however made at the expense of design comprehension and software quality

Ease is designed in a modular fashion to aid evolution, development, addition or enhancement of its services Executive services can be made application specific or target platform specific by adding or modifying modules Simplicity is chosen as a fundamental design principle as it inherently makes *Ease* more predictable, dependable, robust and optimal Refer to the *Ease* User's Guide Appendix A for further details

3.1 Coding conventions of *Ease*

Ease is coded in a mix of ANSI C and assembler The assembler source code is used only where necessary i e for certain kernel operations not facilitated by C and for platform specific hardware interface

All source files are compiled or assembled and their object files archived into the library file `ease lib` This library is then simply linked into an application to utilise the *Ease* software platform

The interface to this library is defined in the API header files `ease h` and `easeinit h` These header files declare all the *Ease* functions and variables

which the application may need

Ease is however made up from a range of interdependent modules The following sections detail the particular conventions and the software practice used in coding *Ease*

3.1.1 Identifiers

All *Ease* functions and global[1] variables are prefixed by `Ease` For example the following declarations are made in the file `kernel h`

```
extern int  EaseClockTick,
extern void EaseScheduleAfterInt(void),
```

This scheme is used to avoid clashes with user applications which may accidentally have variables of the same name as an *Ease* variable *Ease* uses global variables to share certain data between modules

The following definitions are made in `kernel h` to qualify variables declared outside functions

```
#define public
#define private static
```

These definitions are used to limit the scope of identifiers within the source code of *Ease* It is good software practice for a number of reasons to limit the scope of identifiers because only functions which need the identifiers should have access to them The scope of an identifier can be limited in C by using or omitting the `static` qualifier The additional symbolic constants `private` and `public` were added to make the scope of the identifiers clear in coding It would be preferable if the the default scope for C identifiers was `private` but however this is not the case and the source code of *Ease* replaces the omission with the explicit qualifier `public` to clearly indicate that the particular identifier is being consciously and deliberately made public The word `static` has no mnemonic value in the context of limiting scope of identifiers therefore it is replaced in the source code of *Ease* with `private`

```
private int idle_priority,
public  int EaseCurrentPriority,
```

[1]Global in this sense meaning variables visible to any part of the entire program

The variable `idle_priority` is used only by the source module `kernel.c` and is therefore declared as private. The variable `EaseCurrentPriority` needs to be accessed by other *Ease* modules and is therefore declared as public.

The identifiers used in *Ease* are intended to be as descriptive as possible without being too lengthy. If they comprise of more than one word each word is separated either by underscores or by modulating capitalisation between the first and subsequent letters of each word.

3.1.2 Source Modules

All *Ease* C source modules comprise of a C source file and a header file. The source file contains all the functions and data possessed by the module. The header file defines the interface to that module. The source module will at least include its own header and the headers of other modules with which it interacts.

The modules comprise of the following elements:

- Definitions specific to the module
- Public data accessible to other modules
- Functions callable from outside the module
- Private data exclusively used by the module
- Functions exclusively used by the module

There is also another layer from an API (Application Programming Interface) perspective. These are the functions and data accessible to the application. These are defined in the API header files `ease.h` and `easeinit.h` and are not dealt with in this section.

All definitions specific to a module should appear in only one place and may need to be visible to other modules, therefore *Ease* places these definitions in the module's header file. These definitions may be made with for example, the `#define` preprocessor directive or the `typedef` declaration.

The following section of `kernel.h` illustrates this.

```
#define MAX_TASKS 15
#define PRIORITY_LEVELS 8
#define QUANTUM 2

#define TIMER0_INTVEC     0x9

typedef void (*EaseTaskId_t)(void),
```

All data which needs to be accessible to other modules appears in the module's header file. *Ease* header files always use the `extern` storage class specifier to declare the data which the module allows external access. This data is also declared (without the `extern`) as `public` in the C source file.

The module's functions callable from outside the module are also declared in the module's header file using the `extern` storage class specifier. Any module which requires to use these functions simply includes the header file of the appropriate module.

Variables and functions which are used exclusively by the module are present only in the module's source file and do not appear in the module's header. The variables are explicitly declared as private in the module's source file.

3.1.3 Assembler Source Code

The assembler source code is used for operations not facilitated by C. The assembler is used in conjunction with ANSI C code and therefore must not violate the C environment from the application's perspective. This means that it must respect the C register conventions[16].

The assembler functions of *Ease* are also C callable. This means that they have the same stack calling convention as the TMS320C30 C compiler and that all function names are prefixed by an underscore. Their function prototypes are declared in a header file of the same name. *Ease* currently has two assembler files `kextra asm` and `tms_if asm`. `kextra asm` is used for *Ease* specific operations such as context saving and restoring and `tms_if asm` is used for hardware specific peripheral interface.

3.2 Target Platform Of *Ease* Prototype

Ease is developed using the TMS320C30 C compiler and support tools on an MSDOS platform[16][15]

Some of the design of *Ease* is specific to the TMS320C0 microprocessor and to the specific hardware configuration of the particular card used to prototype it The target system is described in chapter 4 1

One of the most basic constraints of a target system is the microprocessor The CPU clock speed is an important factor for the system to meet timing constraints Instruction sets, addressing modes and language support all contribute to have an impact on the software design *Ease* must use assembler routines in conjunction with C to handle features which the C language does not facilitate such as context switching and interrupt enabling and disabling

The hardware elements specific to the TMS320C30 microprocessor which have a direct impact on *Ease* are its two on chip 32-bit timers One of these is used as a system clock for *Ease* to generate its time slicing tick *Ease* employs assembler routines to deal with timer setup

The specific card that was used to prototype *Ease* is the Loughborugh Sound Images TMS320C30 card[17] The hardware elements of this which affect *Ease* are the dual port memory access space (which is used for interface to a PC) and the analog interface The analog interface comprised of two 16-bit digital to analog converters and two 16-bit analog to digital converters One of the TMS320C30 timers is used to trigger conversions for both of the analog to digital converters This constrained *Ease* to use the other timer as a system timer

3.3 *Ease* Kernel Module

This module is the core module of *Ease* It contains the `main` function of the entire program It calls a number of assembler functions from `kextra asm` It deals with the following three activities

- System Initialisation
- Scheduling

- System Timer Services

3.3.1 Task Control Structure

The Scheduling mechanism of *Ease* is based on the way it groups elements of its task control structure The task control structure is defined in `kernel h` and reads as follows

```
typedef void (*EaseTaskId_t)(void),

typedef struct EaseTaskCtrl_s
{
    int    blocked_status,   /* FALSE if not blocked else TRUE */
    int    quantum_tick,
    int    task_sp,
    int    task_id,                /* Integer to identify task */
    EaseTaskId_t root,        /* address of root function of task */
    struct EaseTaskCtrl_s *next_member, /* pointer to next member */
} EaseTaskCtrl ,
```

There are five data fields in the above structure The `blocked_status` field indicates if the task is blocked or ready to run

The task's quantum tick is incremented every time there is a system timer interrupt and the task represented by that structure is currently running If the `quantum_tick` exceeds `QUANTUM` which is defined in the file `kernel h` the quantum tick is reset and the scheduler is called

The `task_id` field is a unique integer to identify the task and is also an index into an array (the kernel's array of tasks) The `root` is a pointer to the root function of the task represented by the structure The root function of a task is similar to the `main` function of standard C program to that task

The `task_sp` is the address of the last location of the task's stack pointer after a context switch *Ease* uses this to restore the task's context

3.3.2 Functions of the Kernel Module

The kernel module performs the initialisation of its task control structure It gets the essential task information from calling the user defined function

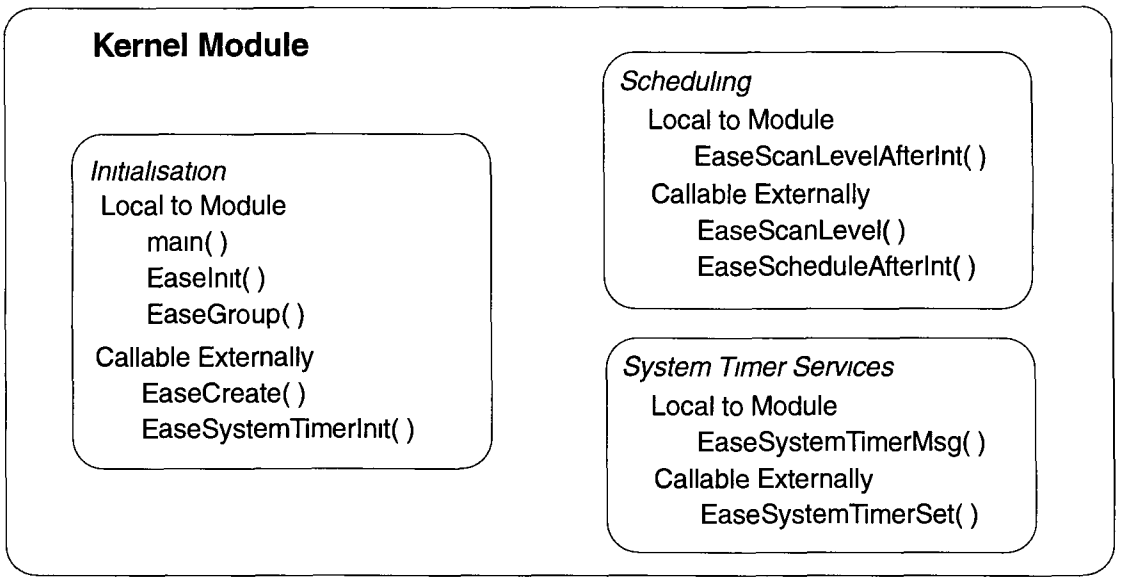


Figure 3 1 The Kernel Module

`EaseForge( )` which in turn calls the kernel function `EaseCreate( )` for each task in the system The function `EaseCreate( )` initialises a data structure for each task These data structures are then grouped into the *Ease* internal management scheme by calling the function `EaseGroup( )` The kernel's initialisation function also calls the intertask communication module's initialisation routine `EaseChannelInit( )` which will be discussed in the following section Finally with all the task structures initialised and grouped the variables `EaseCurrentTask` and `EaseCurrentPriority` are set

The initialisation is completed with *Ease* enabling the interrupt for the system timer and running the first task The first task to run will be a task on the highest priority level The user defined function `EaseForge( )` will also make a call to the kernel module's function `EaseSystemTimerInit( )` The system timer generates the clock ticks for the *Ease* round robin scheduler

The kernel module handles all scheduling after events There are two scheduling functions in the kernel module The simple scheduling case occurs when a task's quantum is expended and is handled by the function `EaseScanLevel( )` In this case the task's context will switch to a task on

the same priority level The more general scheduler is invoked after an event which could make a task of any priority level ready The function to handle this is `EaseScheduleAfterInt( )`

The functions `EaseSystemTimerSet( )` and `EaseSystemTimerMsg( )` handle the kernel's system timer services These functions were added to the kernel after the timer handling module was stabilised and provide timer services from the system timer They will be dealt with in the timer module section

3.3.3 *Ease* On line Task Management

Each task in the system is represented by an `EaseTaskCtrl` structure As the number of tasks is generally fixed in an embedded real time system it was decided that there should be a fixed number of task control structures which could be used by an *Ease* application The kernel module possesses an array of `MAX_TASKS` task control structures This scheme precludes the need for dynamic memory allocation in the kernel module The array also allows quick access to a task structure through knowing the task's `task_Id` field which is an index into this array

Ease manages tasks through grouping its task control structures into closed linked lists of tasks of the same priority It maintains an array of current task pointers to record the current task at each priority level This initialisation is handled in the functions `EaseInit` and `EaseGroup` in the kernel module

Ease also has the following two data arrays to aid real time performance

```
int EaseTaskPr[MAX_TASKS]
int EaseNTasks[PRIORITY_LEVELS],
```

The `EaseTaskPr` is an array storing the task priority for each task The task priority can be accessed knowing the `task_id` The `EaseNTasks` array is used to store the number of tasks at each priority level

The kernel's time slicing scheduler uses these closed linked lists to quickly select the next task to run when a task has expended its quantum This switching is handled by the *Ease* kernel function `EaseScanLevel`

A more general scheduler is required for an event other than a system timer interrupt The is because the event could potentially cause any task in

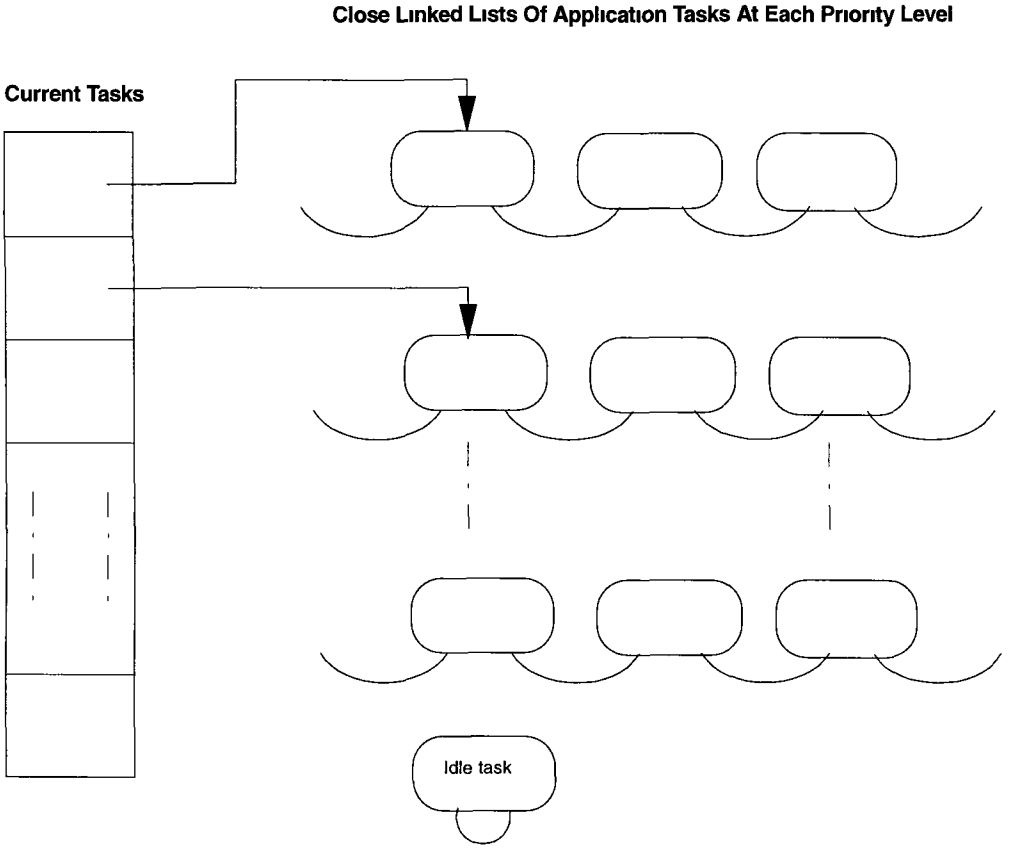


Figure 3 2 *Ease* Task Control Structures

the system to be unblocked including a task at a higher priority than the task currently running The general scheduler must scan through all the tasks of the system starting at the highest priority working down to the lowest

3.4 *Ease* Communication Module

The *Ease* message passing scheme which is used for intertask communication is defined in the `comm` module of *Ease* This is an important module as it not only handles communication but also defines the event interface for later modules which will also use the `comm` module to convert events to messages which are compatible with this scheme

The *Ease* message passing scheme is unbuffered[2] All interrupts are disabled while in the `comm` module as it transfers data between tasks (or rather memory belonging to different tasks) The module consists of three functions

```
void EaseChannelInit( )
int EaseSend(    int dest_Ch,void msg[],
                 int msg_size,
                 EaseTaskId_t * rendezvous_tsk),
int EaseReceive(int src_ch,
                 void msg[ ],
                 int max_msg_size,
                 int *msg_size
                 EaseTaskId_t * rendezvous_tsk),
```

Ease tasks send to and seek messages from *Ease* communication channels Though the scheme is unbuffered there may be a number of tasks blocked waiting on a certain channel These all have to be queued

3.4.1 EaseChannelCtrl Structure

The `EaseChannelCtrl` structure is central to the message passing scheme Each `EaseChannelCtrl` structure represents a single *Ease* communication channel *Ease* maintains an array of these The structure reads as follows

```
struct EaseChanCtrl_s
{
    int source_flag,
    int id_q[MAX_MESSAGES],
    int *msg_q[MAX_MESSAGES],
```

[2]If buffering is required then a task may be created to provide it using this scheme

```
        int size_q[MAX_MESSAGES],
        int head_q,
        int tail_q,
    } EaseChanCtrl,
```

The source_flag field indicates whether the channel is currently a queue of senders or receivers

The id_q[ ] is the queue of task_ids from tasks blocked pending on rendezvous

The msg_q[ ] is an array of pointers to the sender's message or the receiver's message buffer corresponding to the tasks queued at id_q[ ]

The size_q[ ] is an array of sender message sizes or receiver message buffer sizes corresponding to the tasks queued at id_q[ ]

The head_q and tail_q are indexes into the queue arrays indicating the head and tail of the channel's queue

3.4.2 Comm Module Initialisation

The function EaseChannelInit simply sets the head_q equal to the tail_q for each communication channel This indicates that each channel is empty and has no tasks blocked pending rendezvous

3.4.3 Sending

When a task makes a call to EaseSend specifying a channel, there can be one of three outcomes

1 Rendezvous with a receiving task on that channel

2 No rendezvous, task blocked and queued on that channel

3 Call fails through error

If there is a rendezvous, there is a receiving task, with a suitably sized message buffer, blocked and queued on that channel The following happens

- The message is transferred
- The receiver is unblocked

- The receiver is informed of the sender
- The receiver is informed of the actual message size
- The tail of the channel queue is advanced
- The *Ease* general scheduler is called

The sender is allowed to continue and is not blocked The general scheduler will decide whether the receiver or the sender runs next depending on their priorities The reciever is informed of the sender task and actual sender's message size through locations referenced by arguments passed to the receive call

If there is no rendezvous then the sending task has no receiving task to rendezvous with on that channel The following happens

- The channel `source_flag` is set to `TRUE`
- The sender `task_id` is stored on the channel's `id_q`
- The sender message size and pointer is stored on queue
- The sender task is blocked
- The head of the queue is advanced
- The *Ease* general scheduler is called

An error is returned if there are receiving tasks blocked on that channel, but none has a large enough message buffer to take the sender's message

3.4.3 1 Choosing Between Multiple Receivers

The order in which receiver tasks request senders can not be predicted Likewise if a number of tasks are queued up pending rendezvous it can not be predicted which one will rendezvous first The `comm` module of *Ease* will choose the task that has been the longest time in the receiver queue for rendezvous if a sender task becomes available If this sender task has a greater message size than the size argument passed by the receiver task the `comm`

module will scan through the list of pending receivers until it finds a receiver with a suitably sized message buffer If there is not one available it returns an error

3.4.4 Receiving

When a task makes a call to `EaseRecieve` specifying a channel there can be one of three outcomes

1 Rendezvous with a sending task on that channel

2 No Rendezvous, task blocked and queued on that channel

3 Call fails through error

If there is a rendezvous, there is a sending task with a suitably sized message, blocked and queued on that channel The following happens

- The message is transferred
- The sender is unblocked
- The sender is informed of the receiver
- The tail of the channel queue is advanced
- The *Ease* general scheduler is called

The receiver is allowed to continue as normal upon the return from the receive call The general scheduler will decide whether the reciever or the sender runs next depending on their priorities The sender is informed of the receiver task through locations referenced by arguments passed to the send call

If there is no rendezvous then the receiving task has no sending task to rendezvous with on that channel The following happens

- The channel `source_flag` is set to `FALSE`
- The receiver `task_id` is stored on the channels `id_q`
- The receiver message buffer size and pointer is stored on queue

- The receiver task is blocked
- The head of the queue is advanced
- The *Ease* general scheduler is called

An error is returned if there are sending tasks blocked on that channel, but all of them seek to send messages larger than the receiver's message buffer

3.4.4.1 Choosing Between Multiple Senders

The order in which sender tasks request receivers can not be predicted Likewise if a number of tasks are queued up pending rendezvous it can not be predicted which one will rendezvous first The `comm` module of *Ease* will choose the task that has been the longest time in the sender queue for rendezvous if a receiver task becomes available If this sender task has a greater message size than the size argument passed by the receiver task the `comm` module will scan through the list of pending senders until it finds a sender with a suitably sized message If there is not one available it returns an error

3.5 *Ease* Timer Module

The *Ease* timer module provides timer services in the hardware system's application timer which also triggers the A/D converters The module has three functions They are initialisation of the application timer, task interface to the timer or sampler and event handling when an application timer event occurs

The initialisation of the application timer is done at startup by the user defined function `EaseForge` and sets the rate of the application timer and enables its interrupt This is done in the function `EaseApplicationTimerInit`

The functions `EaseApplicationTimerSet` and `EaseSamplerSet` are called by tasks so that the application can avail of application timer services They attach a channel to the timer event, indicate the amount of events before the message is sent and set the mode of the timer service interface as being `ASTABLE` or `MONOSTABLE`

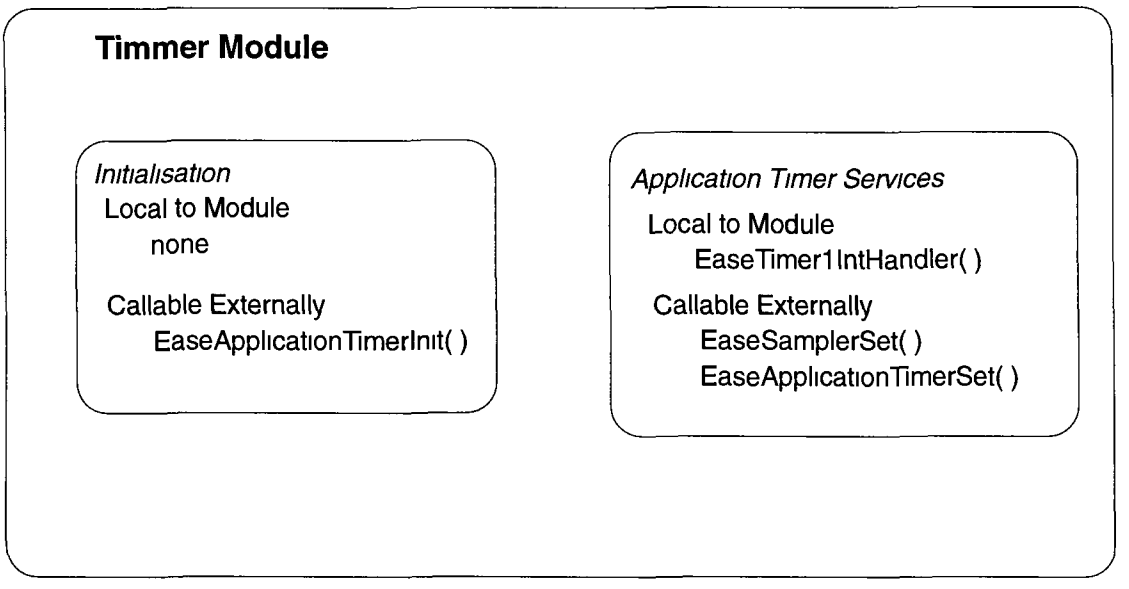


Figure 3 3 The Timer Module

The function `EaseTimer1IntHandler` is called upon an application timer event and effectively emulates a task calling the `EaseSend` function

The Kernel module also has similar services for the system timer As the system timer is needed for scheduling it has the limitation that its base frequency must be chosen to be suitable for *Ease* and not an arbitary value

3.6 Generic Event Handling Modules

Generic events like the *Ease* `Int0` module basically have two functions, one to attach a channel to the interrupt and enable it and another to emulate the `EaseSend` function The function which attaches the channel to the event takes one argument, the channel number This function does the interrupt initialisation and enables the interrupt The function which emulates the `EaseSend` function is at a basic level an interrupt service routine The target platform specific interrupt handler is placed in the file `kextra asm` This then calls the C routine of `Int0` module

3.7 Conclusion

Ease is designed and coded in a modular fashion using ANSI C and assembler This modularity is designed to localise modifications to aid evolution, development, addition or enhancement of its services The assembler code is used only where necessary for operations not supported by C and to perform certain platform specific hardware interface

The software design of *Ease* encourages a structured approach from application programmers The application is divided into a number of dedicated tasks which co-operate with each other in a timely and orderly fashion co-ordinated by *Ease* There is scope for tasks of different priorities and time slicing between tasks of the same priority

Ease employs message passing as a means of intertask communication and synchronisation as both are encapsulated in the same mechanism The message passing is designed in a flexible way over a number of communicating channels which are independent of tasks and facilitate variable sized messages The message passing scheme is expanded to handle external events (interrupts) in a generic way by *Ease* converting them into messages

Chapter 4

Implementing Applications with the Executive

This chapter describes the target platform and the software tools used for prototyping *Ease*. It goes on to describe the implementation of *Ease* from the software design and how the code was prototyped and validated. Once the code of *Ease* was stable, platform timing information was obtained and is shown in section 4.2.3. This chapter concludes describing actual applications which were run with *Ease*, the most notable being the motor modelling of a shock absorber which involved seven concurrent tasks controlling two motors[32].

4.1 The Target System

The target platform for this project uses the TMS320C30 microprocessor. The executive shields the application programmer from excessive knowledge of the hardware through providing its executive services. The following section describes the TMS320C30 as described from the Texas Instruments users guide [14].

4.1.1 TMS320C30

The TMS320C30 is a high performance CMOS 32-bit floating point device in the TMS320 family of digital signal processors. It achieves this performance by implementing many functions in hardware which other microprocessors

implement in software or microcode

This single chip computer system can perform parallel multiply and ALU operations on integer or floating point data within a single cycle The processor also possesses a general purpose register file, program cache, dedicated auxiliary register arithmetic, internal dual access memories, one DMA channel supporting concurrent I/O, and a $60ns$ single cycle execution time

General purpose applications are greatly enhanced by 258K of RAM, multiprocessor interface, internally and externally generated wait states, two external interface ports, two timers, two serial ports, and multiple interrupt structure

The register based architecture lends itself well to implementing high level languages The processor has an associated C compiler and there are numerous software tools for program development

4.1.2 Hardware description

4.1.2.1 Performance

- 60 ns execution time
- 33 3 MFLOPS
- 16 7 MIPS

4.1.2.2 Features

- 4K word single cycle access on chip ROM
- 2K word single cycle access on chip RAM
- 8 extended precision registers
- On chip DMA controller
- Parallel ALU and multiplier operations in single cycle
- Block repeat capability
- Zero overhead loops with single cycle branches

- Interlocked instructions for multiprocessing
- Two on chip serial ports 8/16/24/32 bit transfer
- Two on chip 32-bit timers
- 4 external interrupts

Refer to [14] for further detail on the target microprocessor

4.1.2.3 Software Tools

Texas Instruments have a number of software tools for program development on the C30 The TMS320C30 linker generates object files in a Common Object File Format (abbreviated to COFF) The TMS320C30 linker employs a command file for information on memory configuration of the target hardware platform This command file also allocates *sections* to particular places in the target memory map The linker places code and data from output `obj` files in these sections thereby populating absolute memory without the need for embedding absolute addresses into source files In this way source files can be written independently of the target platform because the `cmd` file is application platform specific For embedded applications the code, reset vector and data constants sections would be placed in ROM through the `cmd` file

The TMS320C30 linker creates executable modules by linking COFF object files The linker allocates sections into the target system's memory, it relocates symbols and sections to assign them to absolute addresses and it resolves undefined external references between input files The linker has a command file associated with it which is used to do the following

- Define a memory model which conforms to the target system memory
- Combine object sections
- To define or redefine global symbols at link time

4.1.3 The TMS320C30 Optimising C Compiler

The processor's C compiler is a full-featured optimising compiler which translates ANSI Standard C to TMS320C30 assembly language The TMS processor uses 32-bit data sizes for floating point and integer values The C compiler also supports large and small memory models[15][16]

4.1.4 The L.S.I. TMS320C30 Card

The specific target platform for prototyping this executive is a development card developed by Loughborough Sound Images Ltd[17] The system consists of a TMS320C30 target system which has a resident monitor program There is a dual port memory interface to a host development PC It has software support for two monitor programs and facilitates assembly and C programming

4.1.4.1 Analog Interface

The card's analog interface consists of two 16-bit analog to digital input channels and two digital to analog output channels The analog interface can operate at sampling rates of up to 200 kHz The input channels include sample and hold amplifiers and both input and output channels are buffered by 5th order Sallen-Key anti-aliasing filters All converters in the interface are triggered by an on board timer or a software trigger Therefore they are not independently triggered channels

The analog input and output channels are accessed by a single 32-bit serial, shift register The input signal is latched to the output upon next interrupt unless the user program intervenes The 32-bit on chip timer has a resolution of $120ns$ hence conversions can take place at very precise intervals

4.2 Executive Implementation

The system specification of chapter 2 describes the executive's requirements and the system design describes how these requirements will be met Once the system specification and design are finalised the next stage of the software

development cycle is the programming phase This phase puts the design in to practice and establishes whether the design can be implemented

4.2.1 Coding

The executive was coded in a modular fashion The main thrust of the coding effort was to develop the executive by incrementally adding features to the code starting with a base line functionality The base line feature of the executive is its ability to multitask by switching processor context between tasks Before any of this work was started time was spent becoming familiar with the target platform and development tools

The code development cycle was as follows

1 Scheduler

- Context switching
- Allocation of task stack and data
- Task control block manipulation
- Time slicing mechanism

2 Inter-task communication

- Message passing mechanism
- Task interface to communication scheme

3 Event Interface

- Event handling mechanism
- Conversion of events to messages
- Task interface to event services

4 System initialisation

- Structured generation of tasks
- Application interface to system initialisation

5 External interface

- Interface specific to target platform
- Structured task interface to service

Each unit of code was developed and tested individually The high level C code was first prototyped on Borland Turbo C++ V1 01 interactive debugging environment before being integrated into the target system This availed of the C debugging tools to thoroughly test the logical correctness of the programs at a high level before cross compiling them The target development degbugging tools only facilitated assembler debugging The assembly routines were tested directly on the target platform through the target platforms monitor program[17] The system design must be clear and precise in order to define the exact specifications for each code unit

The coding of each module involved the following stages

1 Prototype code on PC interactive environment

2 Ensure prototyped code meets design requirements

3 Specify assembly routines necessary or specific to target platforms

4 Develop and test assembly modules (if any) in isolation

5 Integrate high level and assembly code

6 Cross compile to target platform

7 Test feature on target platform

Prototyping the platform independent aspects of a module facilitates rigorous testing of the code with comprehensive debugging tools When one is confident that the code is robust and meets its requirements then the code can be cross compiled to the target platform The prototyped code almost invariably requires simulation code to act as scaffolding to substitute for software or hardware modules not present

The design and the coding experiments isolate the platform specific assembly routines which must be incorporated into the executive These routines can be tested in isolation with the target platform's debugging tool When one is confident that these perform correctly for all inputs then one can integrate them into the module and the module into the executive

4.2.2 Validation

Reliable software is a direct result of a good design process, good software engineering practice and rigorous system testing When each module was finished and passed its unit test it was ready for integration into the executive and validation The goal of testing is to ensure that the software meets it's requirements In real time software there is a large emphasis on the temporal qualities as well as the logical correctness of the system

The code of *Ease* was tested for logical correctness by isolating each function and providing inputs which exercise all flow of control paths within that function This approach minimises the chance that latent errors will pass through the testing stage This technique was adopted for testing *Ease* code as it comprehensively tests the software without being excessive The exercising of all flow of control paths is greatly aided by keeping executive functions concise and clearly defined As interrupts are disabled during kernel operation the validation of logical correctness is free from temporal considerations

An important tool in validation is embedding software tests within functional code for diagnostic purposes These may be added purely for development purposes or as general system integrity tests in the final product

With the executive validated for logical correctness then a test application must be derived to test the executive in a specialised application and to obtain timing information

For example, to test the time slicing module of the executive a number of tasks were created with equal priority These tasks all comprised of a routine which takes a known time to execute and has a counter A break point is placed at the end of the routine of one of the tasks It can be verified that the scheduler was run by the counters of all tasks advancing and through software check points within the scheduler's code The executive's overhead can be calculated through calculating how much longer the tasks took to execute their routine than their stand alone execution times

4.2.3 Platform Timing Information

The amount of op-codes executed by the processor is directly proportional to the amount of time a particular piece of code takes The best case for

this is if each op-code takes one processor clock cycle to execute The clock cycle for the TMS320C30 is 60 nano seconds The source files of *Ease* (both assembler and C) generated code with the following op-codes (counting all branch codes as 4 op-codes)

- Context Switching and scheduling
  - Context Save 40 op-codes
  - Scan Task Control Structure 34 + (14 n) op-codes
  - Context Restore 41 op-codes
- Message Passing
  - Send (with rendezvous) 131+(2n) op-codes (worst case path)
  - Send (no rendezvous) 78 op-codes (worst case path)
  - Receive (with rendezvous) 121+(2n) op-codes (worst case path)
  - Receive (no rendezvous) 81 op-codes (worst case path)

The n for scanning the task control structure is the amount of tasks the scheduler must scan through to find next task to run If the task to run is at the highest priority level for example the worst case for this number is the amount of tasks at that priority level minus one The amount of time spent in a call to `send` or `receive` depends on whether rendezvous is achieved If rendezvous is achieved either call may be responsible for transferring the message The n for send or receive or send is the amount of 32-bit words to be transferred by the executive As the size of messages is variable this can take a variable amount of time to execute

The above data translates into the following times when an op code takes 60 nano seconds

- Context Save, Schedule and Restore 6 900 + (0 840n) μs
- Send (no rendezvous) 7 860 + (0 120n) μs
- Send (with rendezvous) 4 680 μs

- Receive (with rendezvous) 7 260 + (0 120n) μs
- Receive (no rendezvous) 4 86 μs

It was discovered during the testing of *Ease* that these figures are depended on a op-code taking 60 nano seconds This is not the case when memory access has a wait state of a number of clock cycles The above figures were found to be of the order of 60 percent greater if the critical executive data structures were placed in SRAM general memory These figures only held if the critical data structures of *Ease* were stored in the on chip RAM which has a zero wait state The context switching routine also involved `PUSH` and `POP` operations on the stack which required the stacks of the application tasks to be in this zero wait state on chip RAM for the above figures to hold for context storing and saving

4.3 Executive Applications

4.3.1 Analog Signal Display

A primary application was chosen to display two analog input channels on a PC in real time The analog input channels each have a gain applied to them and the modified signal is output to one of two analog output channels The gain and sampling rate for the application is configurable from the PC Although this application is straightforward it incorporates a number of real time issues

- Hardware interface in real time
- Internal co-ordination of data
- On line reconfigurability
- Communication with external computer system

The application is decomposed into four dedicated tasks to perform the various functions These tasks all communicate and synchronise with each other through the *Ease* message passing mechanism The tasks also synchronise with the hardware via *Ease*

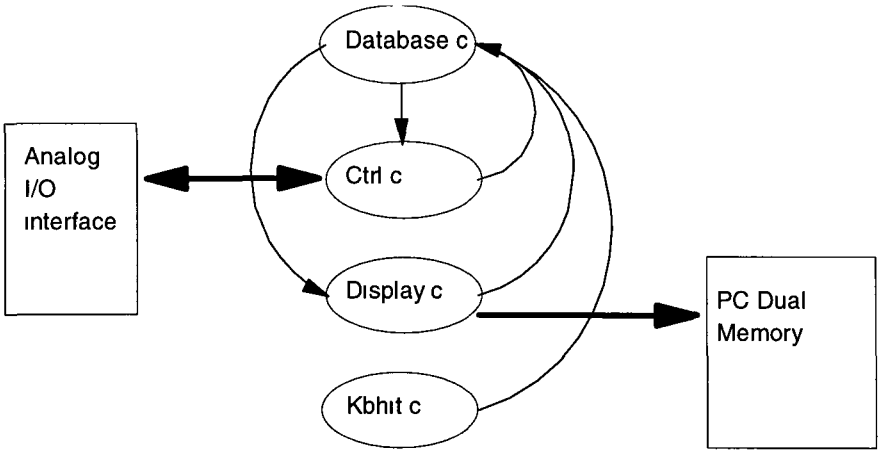


Figure 4 1 Analog Signal Display Application

4.3.1.1 The Database Task

The internal data of the embedded system is managed by the task represented by `database c` data base task which stores the latest parameters and variables Any task that needs information in this task's memory will send a message requesting the information As any task could potentially become blocked waiting for information contained in this task, this task has the highest priority

The core loop of `database c` continuously monitors the `DATA_CH` for a message If the message is a request then it sends the information to the appropriate task from its data, if the message is an update it updates its data accordingly

```
while(TRUE)
{
    receive(DATA_CH,(void *)&msg,sizeof(msg),&msgSize,&Rendezvous),
    switch(msg com)
    {
        case CONTROL_PACKET_REQUEST
            msg parm0=gain0,
            msg parm1=gain1,
```

```
            send(INFO_CH,(void *)&msg,sizeof(msg),&Rendezvous),
            break,
        case DISPLAY_PACKET_REQUEST
            msg parm0=gain0,
            msg parm1=gain1,
            msg int_parm0=signal0,
            msg int_parm1=signal1,
            send(DISPLAY_CH,(void *)&msg,sizeof(msg),&Rendezvous),
            break,
        case IO_UPDATE
            signal0=msg int_parm0,
            signal1=msg int_parm1,
            break,
        case PARAMETER_UPDATE
            gain0=msg parm0,
            gain1=msg parm1,
            break,
    }
}
```

4 3.1 2 The Hardware Interface Task

The hardware interface of the embedded system is handled by the task represented by `ctrl c`, the hardware interface being the access to the analog interface and changing the sample rate of conversions This task also performs the signal processing between analog input and output and updates the data base task with the latest samples This task is placed at a priority below the data base task as it is critical that each sample is stored

4.3.1 3 PC Interface

The transfer of variables to the PC is handled by the task represented by `display c`, which continuously sends a message requesting data and polls the data base task for the latest samples This task also checks the *Ease* I/O flags in the dual memory space between the PC and the target system to see if there is any incoming message from the PC If there is, it activates a PC message handler represented by the task `kbhit c`

4.3.1.4 PC Program

The PC program is primarily a user interface and graphics display program for the application It relays parameters to the target system and presents

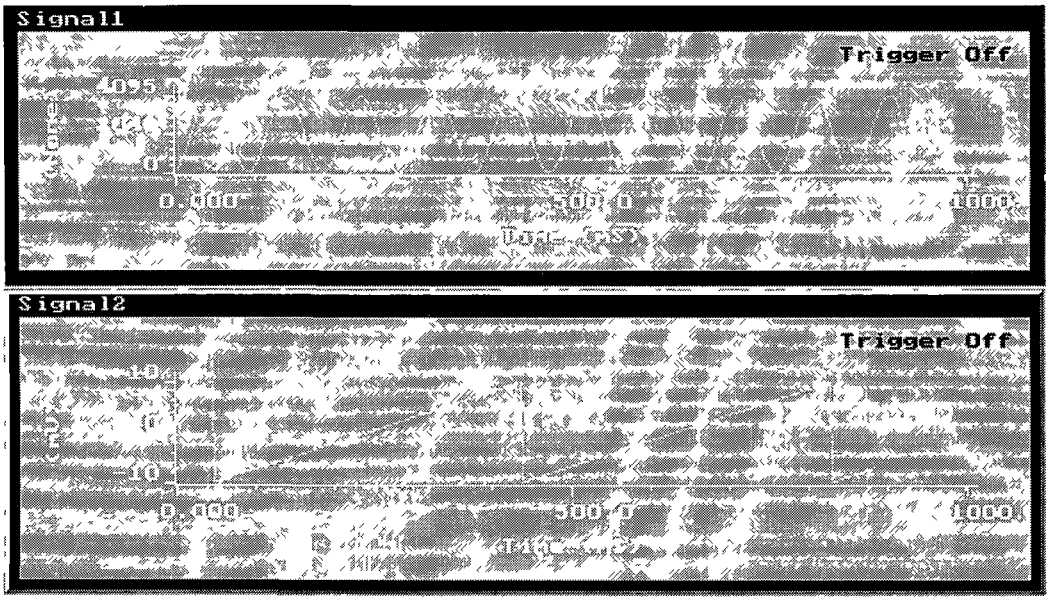


Figure 4 2 Analog Signal Display Screen Dump

data from it in visual form

4.3.2 Motor Control

The motor control application is an extension of the previous application The basic structure of the software is similar with a motor control task that handles signal processing, a data base task that stores the latest control parameters and variables, a display task and a task that polls the PC for commands The application chosen for testing *Ease* is an advanced servo motor application It's background is as follows[32]

> A great deal of interest has developed in the field of active suspensions in recent years especially in the high performance automotive areas[30] Currently the main research is in the hydraulic and electro-hydraulic areas The application considered explores the use of an electrical suspension as a practical alternative The executive structures the application which uses two concurrent motor controllers

4.3.2.1 Target Application System Modelling

The target application comprises of the modelling of a shock absorber using an electro-mechanical system Therefore an analytical model of the mechanical shock is used as the starting block in the control design (Figure 4 3) The reaction force response, F_{act}, is required to simulate that of a mechanical shock absorber (equation 4 1) A linear mass, spring and damper has a rotary equivalent and motor torque is equivalent to linear force By this means disturbance torque, T_d, applied to a rotary motor and controller is treated as being analogous to the linear force, F_{dis}, applied to the mechanical shock A motor controller combination is designed to model the shock absorber The test rig used comprises of two directly coupled permanent magnet DC motors one of which is controlled to act as the load, F_{dis} and the other ,F_{act}, as described above The formulation of the shock absorber takes into account

- the mass of the shock absorber
- the spring and damper constants

$$\frac{X}{F_{dis}} = \frac{1}{M\,s^2 + B\,s + K_s} \tag{4 1}$$

Control: State space control techniques are chosen in this application as they offer the ability to control all states of the system individually in order to achieve the desired response A desired response is produced using a reference model (Figure 4 4) The control law used is a simple feedback of a linear combination of the state variables

Estimation: In order to correctly use the above control method all elements of the state vector equation must be available for feedback purposes To construct the entire state vector a steady state time invariant Kalman filter is used [31] This estimator/filter combines state estimation and sensor filtering The filter contains a model of the motor rig and the disturbance torque The filter is a stochastic filter insofar as the bandwidth of the filter is set by the stochastic properties of the plant and measurement models A

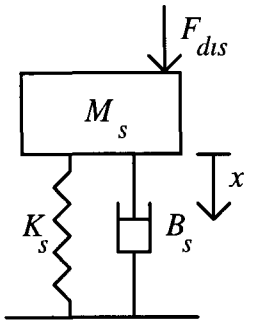


Figure 4 3 Mechanical Shock Absorber Transfer Function Model

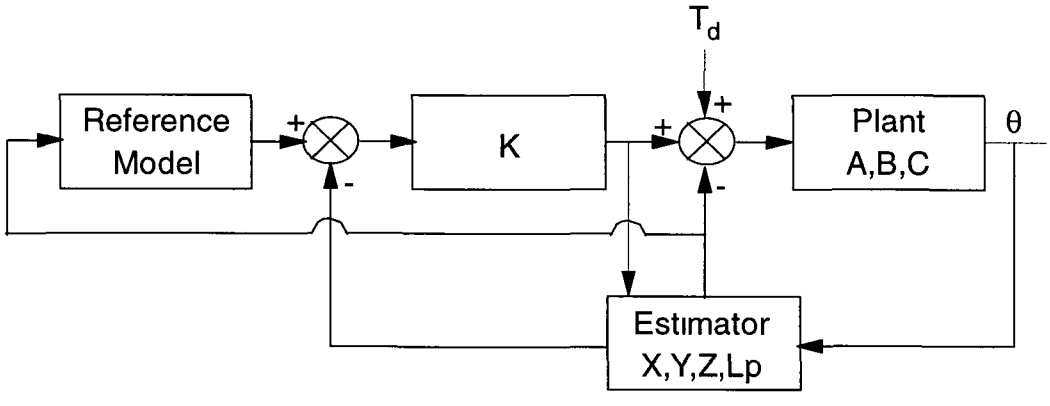


Figure 4 4 Block Diagram of Closed Loop Controlled Plant

steady state implementation is chosen as it is numerically less intensive than the time varying one A state description of the motor model used in the filter is as in equation 4 2

$$\begin{bmatrix} \Theta \\ \dot{\Theta} \\ T_d \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -B_m/J_m & 1/J_m \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Theta \\ \dot{\Theta} \\ T_d \end{bmatrix} + \begin{bmatrix} 0 \\ 1/J_m \\ 0 \end{bmatrix} T_c \qquad (4\ 2)$$

4.3 2.2 Simulation and Implementation

The structure of the motor application is similar to the analog signals application There are more processes running concurrently The motor modelling of a shock absorber involves five independent task objects (figure 4 5) In addition to the controller's task there is a disturbance generation task and an idling task At the highest priority there is the database task which holds within its local memory the latest values of variables, system states and parameters The database task informs the display task of new variables, informs the serial link handler of new system states, informs the controller task of new parameters and accepts updates from any task that modifies or generates data The database task is at the highest priority as any task could potentially be blocked waiting for it to run

The controller task is at the next priority and responds to an *end of conversion* event which indicates that data is ready after a sampling period The other tasks, display, serial link handler and PC message handler involve sending or receiving data from external sources that are not time critical

The executive was found to be sufficiently efficient to run 7 tasks concurrently with on line user interface and concurrent serial interface The motor sampling rate was $4kHz$ The application tasks stacks and the critical data structures of *Ease* did have to be placed in the on chip RAM of the TMS320C30

4.4 Conclusion

Overall *Ease* was found to perform as designed and the applications developed with *Ease* benefited from it *Ease* demonstrated that it has adequate

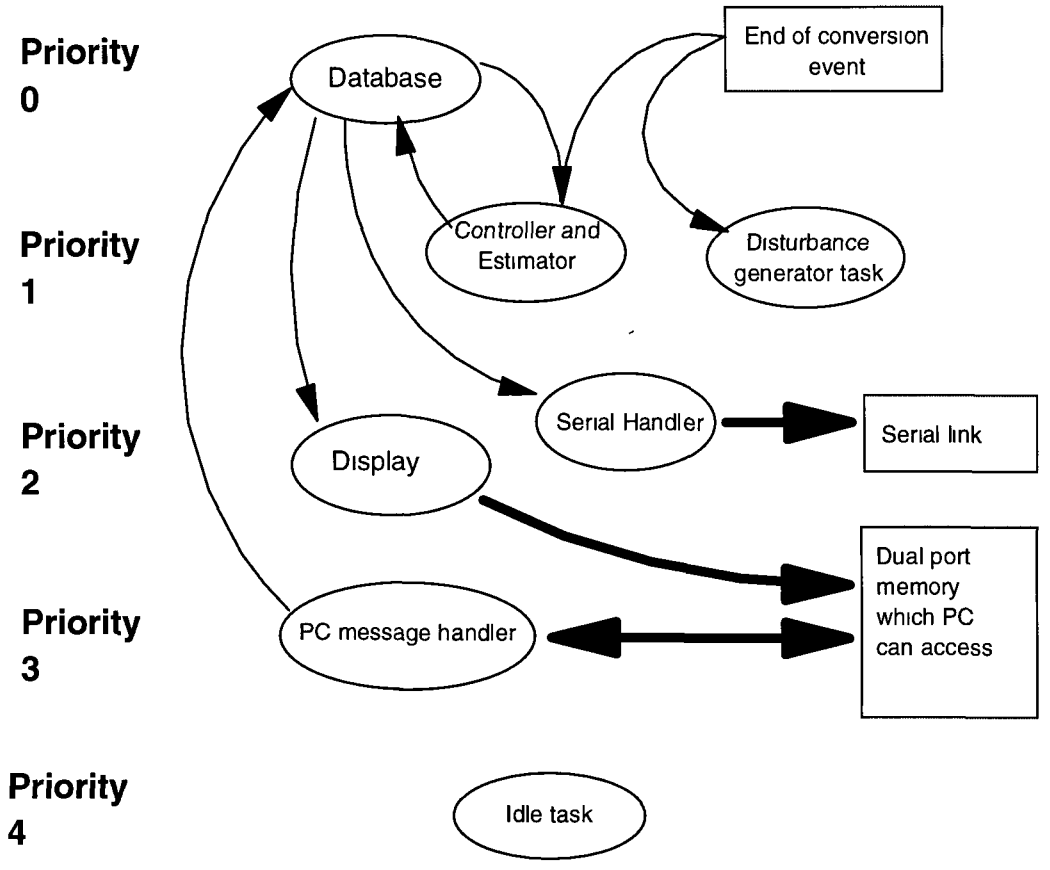


Figure 4 5 Tasks in Motor Application

performance to deal with the motor modelling application which was a good test of the executive

The target system is found to be a good one and a reasonable place where an executive like *Ease* is suited The implementation of the executive benefited from the software practice and a solid design strategy

Chapter 5

Conclusions and Recommendations

5.1 Summary

This thesis describes the development of a real time executive over the stages of specification, design, coding, test and implementation It describes not only the executive but also the software technique involved which is applicable to the development of all systems

The executive facilitates multitasking, inter task communication and synchronisation The executive has been proven to work through a number of sample applications, the most notable being the motor simulation of a shock absorber with an active disturbance generator[32] The multi tasking properties allowed the application to run 7 concurrent tasks each dedicated to a particular function The executive structured the interaction between these tasks The executive overhead was sufficiently low to allow the motor control task and the disturbance task to both run with sampling periods of $4kHz$

5.2 Salient Points

The most important service that the executive provides is that it promotes software quality and maintainability through providing a structured software environment for applications The main thrust of this executive is to enhance software integrity for real time applications

The executive provides synchronisation, scheduling and communication

services It essentially tackles the undesirable aspects of real time programming allowing the application programmer concentrate more on the application and each task to concentrate on the function it is dedicated to

The executive described in this thesis has a number of advantages which differentiate it from existing executives Firstly it is a specialised executive which is targeted at embedded DSP microprocessors It was designed to offer a limited number of sufficient features which makes it efficient, small and easily grasped by the application programmer Furthermore there is access to the source code of the executive which means that the executive is not intrinsically opaque to the application programmer

5.3 Negative Features

The advantages of having multiple independent tasks has the drawback that each one has to have its own stack This leads to both wasteful memory fragmentation between each stack and the potential for stack overflow if not enough stack memory is allocated As each task context is pushed onto its stack by the executive during context switching each stack must be allocated with that much extra stack space It may be argued that in embedded systems memory is so tight and costly that a stack architecture for applications is not practical given the memory constraints The stack runtime memory requirements is largely a unknown quantity forcing application programmers to err on the side of caution Allocating many stacks is a waste of memory because typically they would not all be at their full utilisation and the programmer must allocate for the worst case

For the motor application it was found that the card's DRAM memory was too slow to maintain the application and that all stacks had to be placed in the on chip RAM of the TMS320C30 This point again illustrates the impact of memory issues for real time systems It may be argued that such on chip RAM is a resource designed to be used by an executive

5.3.1 Real Time Stack Integrity

One of the most prominent requirements identified for the executive is a real time stack watch dog This would warn the application programmer if one the task stacks overflows It is very difficult to calculate the run time memory requirements of a developing application and this would be a very useful tool to have

This may be implemented by *Ease* checking the integrity of task stack pointers when ever it gets control through a system call or a timer interrupt The time slicing event for example would give a recurrent opportunity for *Ease* to do this integrity test

A stack fault can not always be detected by checking the stack pointer a particular times For example a stack may overflow into memory it not allocated to it and have returned to its allocated memory space between checks A cathastrophic stack fault may corrupt *Ease* and not allow it to make the check in the first place This stack integrity feature therefore will not catch all stack faults but would get a high percentage of them for a minimum of overhead Unless there is some hardware memory protection by which a task may only use certain memory, stack fault detection be ensured Such hardware facilities are rare in embedded systems

5.4 The Future

The executive was found to have adequate services for most real time applications The multitasking structure directly promotes better applications than unstructured monolithic interrupt driven systems Its API is easy to use and the executive is easy to link into application code

Ease is readily portable to any other TMS320C30 platform There need only to be changes in the command `cmd` file for the particular hardware configuration If the TMS320C30 platform has different hardware resources modules may need to be created to service them Porting *Ease* to a platform that does not use the TMS320C30 requires changes to all non-C code and to modules which are servicing hardware specific functions The general design of *Ease* can still be maintained and reused even if the specific details change

The main thrust of the development of the executive was to develop the executive in a modular fashion, adding in features as modules were developed and integrated This development approach provides ample scope for system improvement The modifications are localised to each module

Appendix A

Ease User's Guide

A.1 Introducing *Ease*

Ease is a real-time multitasking *E*mbedded software *A*pplication and *S*ystems *E*xecutive targeted at DSP platforms The digital computer in a real-time embedded system controls a process by receiving data, processing it and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time The computer is essentially within the control loop and its responsibilities in that role are its primary functions Synchronisation, scheduling and communication between the different components of real-time software in a reliable, timely and predictable fashion places great demands on the software

Ease provides an application software interface to the underlying hardware and encourages a structured approach from application programmers which enhances software integrity and maintainability in a potentially chaotic real-time environment The confidence afforded by *Ease* is paid for by a small percentage of CPU processing power and a larger response time to external events than an unstructured, monolithic, interrupt driven system The focus of *Ease* is to tackle the undesirable aspects of real-time programming and device dependent issues thereby allowing the application programmer to concentrate more on the application

Ease has a number of advantages which differentiate it from existing executives Firstly it is a specialised executive which is targeted at DSP microprocessors It was designed as a low cost executive which offers a limited

number of sufficient features which makes it efficient, small and easily grasped by the application programmer Furthermore there is access to the source code of *Ease* which means that the executive is not intrinsically opaque to the application programmer, as many commercial products are

A.2 Features of *Ease*

Ease is a software platform which facilitates programming of concurrent application *tasks* *Ease* is designed for embedded systems and therefore is a minimal kernel designed to be fast and efficient to reduce overhead and meet timing constrains but not at the expense of design comprehension *Ease* is designed in a modular fashion to aid evolution, development, addition or enhancement of its services Executive services can be made application specific or target platform specific by adding or modifying modules Simplicity is chosen as a fundamental design principle as it inherently makes *Ease* more predictable, dependable and optimal by not allowing unwieldy complexity to creep in

The following are the core services offered with *Ease*

- Pre-emptive event driven scheduling of application tasks
- Synchronisation and communication facilities between application tasks
- Consistent application interface to internal and external events
- Executive support for object oriented techniques

The overall user application is made up of a number of tasks which cooperate with each other in a timely and orderly fashion coordinated by *Ease* *Ease* facilitates and encourages object oriented techniques for design and implementation of application programs Each task is its own entity with it's own data, code and stack and essentially constitutes an object Each task performs services while communication with other tasks is carried out via message passing The form of each task is initialisation followed by an endless loop which typically would have some interface to an internal or external event

A.2.1 Scheduling with *Ease*

The fundamental uniprocessor method for introducing concurrency involves pseudo parallelism This is achieved by switching processor context between independent task objects

Task scheduling is conducted on a priority basis with a time slicing scheme for tasks of equal priority The executive supports static process priority This approach was adopted because the provision for dynamic process priorities may obscure application bugs The scheduler is run upon a scheduling event which may be a hardware interrupt or a software trap Scheduling is guaranteed at a minimal level by a special *clock* interrupt given by a timer which *Ease* uses as a system clock The actual scheduling mechanism is designed to be as fair as possible without excessive overhead The scheduling mechanism does the following on a scheduling event

- Make a limited context switch so the scheduler can run
- Run scheduler to decide which ready task to select on the basis of the relevant states of tasks within the application
- Update task information structure on the basis of the scheduler's decision
- If the same task is to be run restore it
- If another task is to be run do a full context switch

A.2.2 Synchronisation and Communication with *Ease*

Ease employs message passing as a means of inter-task communication and by definition synchronisation is achieved through a rendezvous scheme All communication and event handling is conducted via *Ease* All messages are passed in a call to either `EaseSend( )` or `EaseReceive( )` In that call the task specifies which one of a number of *channels* the message will go to

The calls may be paraphrased as

Send a message to a potential receiver seeking a message from the specified channel number

and

Receive a message from a potential sender sending a message on the specified channel number

A sending task is *blocked* until a receiver is present to take its message If there are multiple readers or writers then the identities of the blocked tasks are recorded until each blocked task has a rendezvous partner *Ease* does not guarantee which of the blocked tasks will rendezvous first

Ease converts external events (interrupts) to messages which the tasks can synchronise with through a *block on receive* mechanism If there is a task waiting for the event then *Ease* sends a message to inform that task that the event has occurred If there is no task waiting then the executive records a lost interrupt and continues on Lost interrupts usually indicate a pathological error in the application's timing or that the processor is too slow for the application

A.3 Working with *Ease*

Tasks access *Ease* services through C function calls As a result of this tasks must be written in a C or a C callable assembler if they wish to avail of any communication or synchronisation services *Ease* provides

A.3.1 Naming Conventions used with *Ease*

All *Ease* functions, type definitions and application wide global variable identifiers have `Ease` prepended to them *Ease* uses this convention to keep *Ease* identifiers distinct from user application identifiers *Ease* can not predict the results of an application programmer generating identifiers which start with the characters `Ease`

A.3.2 Task generation with *Ease*

Task is the name given to a separate concurrent process within an *Ease* application Each task possesses its own run time stack The root function of a task operates like an autonomous concurrent `main( )`[1] function and is initiated at run time by *Ease* The root function's prototype must be of the form `void` *task_name*`(void)` In the concurrent environment functions used by more than one task must be re-entrant so static data is not corrupted Therefore the application programmer must ensure that there is only one task using any non re-entrant function at a time

Ease requires application programmers to write an application specific function called `EaseForge( )` This function consists of calls to set timer frequencies and a series of calls to `EaseCreate( )`[2] to inform *Ease* of the names of the root functions, the priorities and stack allocations of each task in the application

The priority convention in *Ease* defines the highest priority as zero There may up to seven priority levels[3] In the function `EaseForge( )` there must be at least one task created with priority zero and at least one task created in each priority level down to the lowest priority in use

[1]The actual `main( )` function is reserved for use within the *Ease* kernel
[2]Prototype in `easeinit h`
[3]This limit may be altered by changing the source files and recompiling

Here is an sample of how to use `EaseForge( )`

```
/*

File   Forge c

*/

#include"easeinit h"
#include"motorapp h"

void EaseForge(void)
{
    EaseSystemTimerInit(5000),
    EaseApplicationTimerInit(4500),

    EaseCreate(motor_db,      0, 0x800),
    EaseCreate(control_motor, 1, 0x300),
    EaseCreate(display,       2, 0x200),
    EaseCreate(pc_message,    2, 0x500),
}
```

The first argument to `EaseCreate( )` is a pointer to the root function of a task The second argument is the priority level of the task The third argument is the stack space to be allocated for that task by *Ease* The application programmer must be careful when choosing this number If the stack allocated is too small then the task stack may overrun and corrupt data, on the other hand if the sum of the stack allocations are too great then there may not be enough physical memory The stack must be able to accommodate all the local variables of the root task plus those of any functions which are subsequently called to the deepest nested level and must cater for memory taken by the actual parameters passed on the stack

The above `EaseForge( )` function tells *Ease* that the application consists of four tasks The `#include` file `motorapp h` contains among other things the prototypes of the root functions

A.3.3 Services of *Ease*

The services of *Ease* are accessed through direct C function calls An application compilation unit must `#include` the file `ease h` to give the compiler

information on the service function prototypes

The following are prototypes of *Ease* functions

Called from application tasks (prototypes in `ease h`)

```
int EaseReceive(int src_ch,
                void * msg,
                int max_msg_size,int* msg_size,
                EaseTaskId_t* rendezvous_tsk),

int EaseSend(   int dst_ch,
                void * msg,
                int msg_size,
                EaseTaskId_t* rendezvous_tsk),

int EaseApplicationTimerSet(int ticks,
                            int channel,
                            int mode),

int EaseSamplerSet( int ticks,
                    int channel,
                    int mode),

int EaseSystemTimerSet( int ticks,
                        int channel,
                        int mode_c),

int EaseInt0Init(int channel),
```

Called in `EaseForge( )` (prototypes in `easeinit h`)

```
void EaseSystemTimerInit(int freq),

void EaseApplicationTimerInit(int freq),

void EaseCreate(EaseTaskId_t function,
                int priority,
                int stack_alloc),
```

The most fundamental functions are `EaseSend( )` and `EaseReceive( )` as they are central to task communication and synchronisation The above functions are described in detail in Section A 8 and Section A 9

A.4 Current platform of *Ease*

Ease is designed in a modular fashion using C where possible and assembly language where necessary so as to make it as portable as possible Although it is desirable that an executive would be portable, it has inherent machine dependent modules and has to be developed on a physical hardware platform

The target platform chosen for the first prototype of *Ease* is one using the TMS320C30 DSP microprocessor This was chosen for its performance, functionality and the readily available C compiler made by Texas Instruments More specifically the physical platform is a TMS320C30 micro-controller developed by Loughborough Sound Images Ltd The software platform is the Texas Instruments TMS320C30 microprocessor development system

A.4.1 TMS320C30 Command files

The TMS320C30 linker employs a command file for information on memory configuration of the target hardware platform This command file also allocates *sections* to particular places in the target memory map The linker places code and data from output `obj` files in these sections thereby populating absolute memory without the need for embedding absolute addresses into source files In this way source files can be written independently of the target platform because the `cmd` file is application platform specific For embedded applications the code, reset vector and data constants sections would be placed in ROM through the `cmd` file

A.4.2 Platform specific PC interface

The LSI board used for prototyping *Ease* shares a dual port memory area with an IBM PC thereby facilitating two way on-line communication PC to DSP interface requires programs running on both the PC and the DSP The PC program was designed and tested as an MSDos application using Borland Turbo C++ version 1 01 *Ease* provides prototypes for special functions and information for this platform specific interface in an `#include` file called `dsp_if h` which is included by both the PC resident and DSP resident programs This platform specific interface is described in detail in section A 10

A.4.3 Platform timing Information

On the LSI card the executive switches context in $10\mu s$ while time-slicing, it passes messages in $38\mu s$ and responds to interrupts in $8\mu s$ Only tasks on the highest priority level can have that response guaranteed

A.5 Mechanisms of *Ease*

A.5.1 *Ease* error handling

Ease indicates errors through leaving an error message string at a specific location in memory This location can also be accessed by application tasks through the pointer `EaseErrorMessage` which is declared in the *Ease* include file `ease h` An application task detecting a serious error can copy its error message string to `EaseErrorMessage` *Ease* also gives application tasks access to special variables through the include file `ease h` which gives information on interrupts which are lost through no task being ready to respond to them

The memory location which `EaseErrorMessage` points to is chosen to be a location which an external computer system can access The obvious choice for this location on the particular platform which the prototype *Ease* was developed was in the dual port memory space

The following are the *Ease* error messages

- `Too Many Tasks created in EaseForge` This error message indicates that there are too many tasks created for the particular *Ease* compilation
- `Number of task priority levels exceeds max in EaseForge` This error message indicates that there are too many priority levels for the particular *Ease* compilation
- `Ease System timer not initialised` This error message indicates that the *Ease* system timer is not set and *Ease* cannot function
- `Priority Rules not Respected in EaseForge` This error message indicates that there is a gap in the priority levels of the created tasks

There should be at least one task in each priority level

- `Illegal Exit form root task` *task number* This error message indicates that there is a return from a root task function indicated by task number Task number is derived from the order in which tasks are created in the *Ease* initialisation function `EaseForge( )` The first task created in `EaseForge( )` is numbered one

The first two messages indicate that the limits set in the Ease library `ease lib` are exceeded If the application programmer wishes to extend these limits the *Ease* source module `kernel c` must be recompiled with the appropriate changes made to the symbolic constants defined in the *Ease* include file `kernel h` *Ease* will not initiate any tasks or enable any interrupts if any of the first four conditions arise and will sit in a tight loop

A.6 *Ease* Timers

Ease timers generate events at a rate set in the *Ease* application specific initialisation function `EaseForge( )` through calls to `EaseSystemTimerInit( )` and `EaseApplicationTimerInit( )` The prototypes of these functions are declared in the *Ease* include file `Easeinit h` The periods set by these calls are the smallest time divisions with which *Ease* can provide timer services Application tasks can avail of timer services at discrete submultiples of these events

The timers are only set once and are only set before task initiation because timers being reset on-line will affect the realtime software integrity

An application task requests *Ease* to send a message on a specific inter-task communication channel to mark the occurrence of a specific number of timer events *Ease* provides these timer services in an astable or monostable mode The prototypes to set timers are declared in the *Ease* include file `ease h`

Application timers which generate triggers for analogue samplers constitute a special case of *Ease* timer services *Ease* sends a message on the conclusion of each sampling period which is marked by an end of conversion event

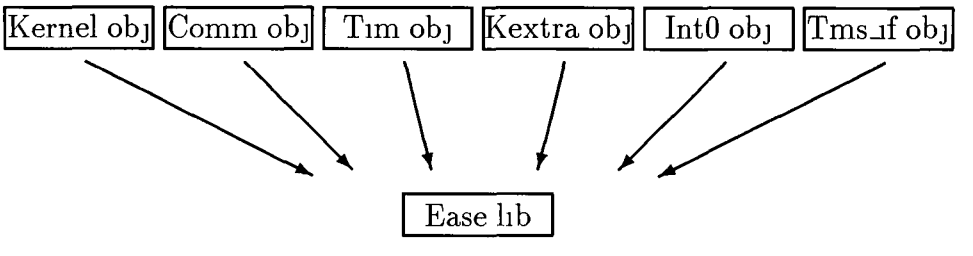
A.7 Directory Organisation of *Ease*

The release version of *Ease* has files in six directories, they are as follows

- `Src` *Ease* source modules and library file `ease lib`
- `Include` *Ease* include files
- `Bin` PC batch and executable files to aid *Ease*
- `Skeleton` Skeletal *Ease* files upon which applications can be built
- `Apps` Sample *Ease* application
- `Motor` Sample motor application with *Ease*
- `Doc` User documentation

A.7.1 Source Files of *Ease*

The source files of *Ease* are compiled, linked and archived into a library file called `ease lib` which the *Ease* application tasks may access through the *Ease* include file `ease h` The *Ease* library file `ease lib` is target platform specific and the source files of *Ease* may have to be recompiled and archived for each particular target platform system In this way the *Ease* application tasks are target platform independent The only application specific function required by *Ease* is the initialisation function `EaseForge( )` The following figure shows the *Ease* modules which constitute the initial version of *Ease*



The above modules have the following purposes

- `Kernel obj` This module handles task scheduling, initialisation and system timer services

- `Kextra obj` This module contains services for the *Ease* source modules which have to be written in written in assembly language
- `Comm obj` This module handles inter-task communication
- `Tim obj` This module handles application timer services
- `Int0 obj` This module deals with interrupts on the external interrupt line 0
- `Tms_if obj` This module deals with data exchange via a dual port memory system

The most target platform specific file on a TMS320C30 system is the *Ease* linker command file `ease cmd` This directs the TMS320C30 linker to place the components of object files in appropriate sections of physical memory

A.7.2 Include files

The *Ease* `include` directory has files which are to be included by application tasks compiled under *Ease* and by the compilation unit which contains the *Ease* initialisation function `EaseForge( )` It contains three files as follows

- `Easeinit h` This file is included for use by the *Ease* initialisation function `EaseForge( )` It declares the prototypes for task creation and timer initialisation
- `Ease h` This is included directly by an application task's compilation units It defines *Ease* symbolic constants, declares function prototypes for *Ease* services and declares application wide *Ease* global variables
- `Dsp_if h` This include file is the most basic include file for PC to DSP data interchange This contains symbolic constants referencing specific locations in the dual port memory interface which will be different for each applications interface The file is therefore usually included as a local include file

A.8 Prototypes of Easeinit.h

This file is used to declare the task creation and timer initialisation function prototypes and should only be called from the compilation unit containing the *Ease* application specific initialisation function `EaseForge( )` The following prototypes are declared by `Easeinit h`

- `EaseCreate( )`
- `EaseSystemTimerInit( )`
- `EaseApplicationTimerInit( )`

The object files which contain these functions are archived into the *Ease* library file `Ease lib`

A.8.1 EaseCreate( )

Prototype

```
void EaseCreate(EaseTaskId_t function,
                int priority,
                int stack_alloc),
```

Arguments

- `EaseTaskId_t function` This indicates to *Ease* the function which is to be made a root function of an *Ease* task
- `int priority` This is the priority level that the created task is to be placed at
- `int stack_alloc` This is the stack allocation for the created task m 32-bit words

Description This function is used to register and initiate a specific *Ease* task The call informs *Ease* of the details of the task's priority, stack allocation and the task's root function identification It is called from the application specific *Ease* initialisation routine `EaseForge( )`

Constraints

- `EaseTaskId_t function` This can be any C function whose prototype is `void` *function_name*`(void)`
- `int priority` If the priority level argument is not valid then an error message will be generated and *Ease* will not initiate any tasks
- `int stack_alloc` *Ease* does not conduct on-line stack monitoring `stack_alloc` must be large enough to accommodate all the local variables of the root task plus those of any functions which are subsequently called to the deepest nested level and must cater for memory taken by the actual parameters passed on the stack If the sum of task stack allocations within an application is too large then there may not be enough physical memory to accommodate them
- The onus is on the application developer to ensure that this function is only called from the *Ease* application specific function `EaseForge( )` *Ease* will behave unpredictably if this is not adhered to

A.8.2 EaseSystemTimerInit( )

Prototype

```
void EaseSystemTimerInit(int freq),
```

Arguments

- `int freq` This number is the frequency at which system clock is to be set at in hertz

Description This call initializes the system timer The system timer is used by *Ease* for time slicing and for application task timer services The frequency of the system timer is set before any task is initiated The period set by `freq` is the smallest time division between system timer events *Ease* allows application tasks to use events for timer services as detailed in section A 9 3

Constraints

- The onus is on the application developer to ensure that this function is only called from the *Ease* application specific function `EaseForge( )` The behaviour of *Ease* will be unpredictable if this is not adhered to

A.8.3 EaseApplicationTimerInit( )

Prototype

```
void EaseApplicationTimerInit(int freq),
```

Arguments

- `int freq` This number is the frequency at which the application timer is to be set at in hertz

Description This call initializes the application timer *Ease* uses this timer exclusively for application task timer services The frequency of this timer is set before any task is initiated The period set by `freq` is the smallest time division between application timer events *Ease* allows application tasks to use events for timer serices as detailed in section A 9 4

Constraints

- The onus is on the application developer to ensure that this function is only called from the *Ease* application specific function `EaseForge( )` The behavour of *Ease* will be unpredictable if this is not adhered to

A.9 Prototypes of Ease.h

The file `Ease h` is used to declare the *Ease* service calls for use by application tasks The following prototypes are declared by `Ease h`

- `EaseReceive( )`
- `EaseSend( )`
- `EaseSystemTimerSet( )`
- `EaseApplicationTimerSet( )`
- `EaseSamplerSet( )`
- `EaseInt0Init( )`

The object files which contain these functions are archived into the *Ease* library file `Ease lib`

A.9.1 EaseReceive( )

Prototype

```
int EaseReceive(int src_ch,
                void * msg,
                int max_msg_size,
                int* msg_size,
                EaseTaskId_t* rendezvous_tsk),
```

Arguments

- `int src_ch` This is a positive integer to identify the source inter-task communication channel which this call seeks to receive a message from
- `void * msg` This is a pointer which indicates to *Ease* where a message will be passed in the receiving task's object memory The type is `void *` to make it a generic pointer type
- `int max_msg_size` This informs *Ease* of the largest message size, in 32-bit words that the receiving task can receive
- `int* msg_size` This returns the actual size of the message received in 32-bit words
- `EaseTaskId_t* rendezvous_tsk` This returns the identifier of the root function of the sending rendezvous partner

Return Value

- `int` The non zero return values following are symbolic constants denoting error messages and are defined in the *Ease* include file `ease.h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `src_ch` is not a defined *Ease* inter-task communication channel
  - `MSG_TOO_LARGE_FOR_RECEIVER` is returned if the receiving task tries to rendezvous with a blocked sending task whose message size is greater than `max_msg_size`

Description In this call a task tries to obtain a message from the *Ease* inter-task communication channel indicated by `src_ch` and blocks if none is available. It will remain blocked until a sending task sends on that channel with a message not bigger than `max_msg_size`

Constraints

- `int src_ch` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

- `void * msg` The `msg` argument is designed to be a pointer to data types within a task's local or static global memory The behavour of *Ease* will be indeterminate if `msg` is a pointer to anything outside a task's private memory space

- `int max_msg_size` This limit is the responsibility of the application developer who must ensure that the space allocated for an incoming message is actually `max_msg_size` If this number is larger than the space actually allocated, data is liable to be corrupted with unpredictable results Typically a `sizeof( )` macro will ensure that this will not occur

A.9.2 EaseSend( )

Prototype

```
int EaseSend(   int dst_ch,
                void * msg[ ],
                int msg_size,
                EaseTaskId_t* rendezvous_tsk),
```

Arguments

- `int dst_ch` This is a positive integer to identify the destination inter-task communication channel, over which this call seeks to send a message
- `void * msg` This is a pointer which indicates to *Ease* where in the sending task object memory the message to be sent exists
- `int msg_size` This indicates to *Ease* the actual size of the message to be relayed in 32-bit words
- `EaseTaskId_t* rendezvous_tsk` This returns the identifier of the root function of the receiving rendezvous partner

Return Value

- `int` The non zero return values following are symbolic constants denoting error messages and are defined in the *Ease* include file `ease.h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `dst_ch` is not a defined *Ease* inter-task communication channel
  - `MSG_TOO_LARGE_FOR_RECEIVER` is returned if the sending task tries to rendezvous with a blocked receiver task who cannot receive a message of size `msg_size`

Description In this call a task tries to send a message on the *Ease* inter-task communication channel indicated by `dst_ch` and blocks if there is no receiver waiting It will remain blocked until a receiving task seeks a message on that channel with a maximum message size greater than or equal to than `msg_size`

Constraints

- `int dst_ch` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

- `void * msg` The `msg` argument is designed to be a pointer to data types within a task's local or static global memory The behavour of *Ease* will be indeterminate if `msg` is a pointer to anything outside a task's private memory space

- `int msg_size` The onus is on the application developer who must ensure that this is the actual size of the message to be sent If the number is too small then an incomplete message will be sent If the number is too large then unspecified data will be appended to the end of the message Typically a `sizeof( )` macro will ensure that these scenarios will not occur

A.9.3 EaseSystemTimerSet( )

Prototype

```
int EaseSystemTimerSet( int ticks,
                        int channel,
                        int mode),
```

Arguments

- `int ticks` This is the number of system timer events an application task wishes to wait before *Ease* indicates a timing event
- `int channel` This is a positive integer which indicates to *Ease* the inter-task communication channel over which a message is to be relayed indicating the occurrence of `ticks` number of system timer events
- `int mode` This informs *Ease* of the mode required from the system timer The timer may be set in a `MONOSTABLE` or `ASTABLE` mode

Return Value

- `int` The non zero return values following are symbolic constants denoting irregularities and are defined in the *Ease* include file `ease h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `channel` is not a defined *Ease* inter-task communication channel
  - `RESET_WHILE_ACTIVE` is returned if the system timer is already servicing a timing request when the current call reset it
  - `INCORRECT_TIMER_MODE` is returned if the `mode` argument is not one of the symbolic constants `MONOSTABLE` or `ASTABLE` defined in the *Ease* include file `ease h`

Description In this call an application task requests timer services from the *Ease* system timer The system timer generates events at a rate defined by `EaseSystemTimerInit( )` This requests *Ease* to send a message indicating that `ticks` number of system timer events have occurred meaning that a specific period of time has elapsed This message is sent on the *Ease* inter-task communication channel indicated by `channel` The message will be sent repeatedly every `ticks` system timer events if the `mode` is `ASTABLE` If the `mode` is `MONOSTABLE` is the message will be sent once after `ticks` system timer events has elapsed

The message consists of a integer array of two 32-bit words, the first word being `SYSTEM_TIMER_MSG` which is a symbolic constant defined in `ease h` and the second being the number of system timer events that happened since system timer initialisation by the function `EaseSystemTimerInit( )`

This call will override any previous system timer request which has not expired

Constraints

- `int channel` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

- `int mode` This must be one of the symbolic constants `MONOSTABLE` or `ASTABLE` defined in `ease h`

A.9.4 EaseApplicationTimerSet( )

Prototype

```
int EaseApplicationTimerSet(int ticks,
                            int channel,
                            int mode),
```

Arguments

- `int ticks` This is the number of application timer events an application task wishes to wait before *Ease* indicates a timing event
- `int channel` This is a positive integer which indicates to *Ease* the inter-task communication channel over which a message is to be relayed indicating the occurrance of `ticks` number of application timer events
- `int mode` This informs *Ease* of the mode required from the system timer The timer may be set in a `MONOSTABLE` or `ASTABLE` mode

Return Value

- `int` The non zero return values following are symbolic constants denoting irregularities and are defined in the *Ease* include file `ease h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `channel` is not a defined *Ease* inter-task communication channel
  - `RESET_WHILE_ACTIVE` is returned if the system timer is already servicing a timing request when the current call reset it
  - `INCORRECT_TIMER_MODE` is returned if the `mode` argument is not one of the symbolic constants `MONOSTABLE` or `ASTABLE` defined in the *Ease* include file `ease h`

Description In this call an application task requests timer services from the *Ease* application timer The application timer generates events at a rate defined in `EaseApplicationTimerInit( )` This requests *Ease* to send a message indicating that `ticks` number of application timer events have occurred meaning that a specific period of time has elapsed This message is sent on the *Ease* inter-task communication channel indicated by `channel` The message will be sent repeatedly every `ticks` system timer events if the `mode` is `ASTABLE` If the `mode` is `MONOSTABLE` is the message will be sent once after `ticks` system timer events has elapsed

The message consists of a integer array of two 32-bit words, the first word being `APPLICATION_TIMER_MSG` which is a symbolic constant defined in `ease h` and the second being the number of application timer events that happened since application timer initialisation by the *Ease* function `EaseApplicationTimerInit( )`

This call will override any previous application timer request which has not expired

Constraints

- `int channel` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

- `int mode` This must be one of the symbolic constants `MONOSTABLE` or `ASTABLE` defined in `ease h`

A.9.5 EaseSamplerSet( )

Prototype

```
int EaseSamplerSet(int channel),
```

Arguments

- `int channel` This is the *Ease* inter-task communication channel over which a message will be passed by *Ease* indicating a sampling event

Return Value

- `int` The non zero return values following are symbolic constants denoting irregularities and are defined in the *Ease* include file `ease h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `channel` is not a defined *Ease* inter-task communication channel
  - `RESET_WHILE_ACTIVE` is returned if the application timer driving controlling the sampler is already servicing a timing request when the current call reset it

Description In this call an application task requests *Ease* to generate messages at an end of conversion event from the target system's analog interface Sampling events are triggered by the application timer which generates triggering pulses at a rate defined in `EaseApplicationTimerInit( )` This message is sent on the *Ease* inter-task communication channel indicated by `channel` at each sampling instant

The message consists of a integer array of two 32-bit words, the first word being `END_OF_CONVERSION_MSG` which is a symbolic constant defined in `ease h` and the second being the number of application sampling events that happened since application timer initialisation by the *Ease* function `EaseApplicationTimerInit( )`

This call will override any previous application timer request set by `EaseApplicationTimerSet` which has not expired

Constraints

- `int channel` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

A.9.6 EaseInt0Init( )

Prototype

```
int EaseInt0Init(int channel),
```

Arguments

- `int channel` This is the *Ease* inter-task communication channel over which a message will be passed by *Ease* indicating an INT0 event

Return Value

- `int` The non zero return values following are symbolic constants denoting error messages and are defined in the *Ease* include file `ease h`
  - A zero is returned if operation is successful
  - `NOT_VALID_CHANNEL` is returned if `channel` is not a defined *Ease* inter-task communication channel

Description In this call an application task requests *Ease* to generate messages at an INT0 event, which is triggered by an external interrupt on the INT0 line This message is sent on the *Ease* inter-task communication channel indicated by `channel` at each event instant

The message consists of a integer array of two 32-bit words, the first word being `INT0_MSG` which is a symbolic constant defined in `ease h` and the second being the number of INT0 events that happened since this call enabled the INT0 external interrupt line

Constraints

- `int channel` This must be a positive integer denoting a defined *Ease* inter-task communication channel Valid channels are in the range zero to the symbolic constant `CHANNELS` which is a limit defined in the *Ease* inter-task communication source module by the include file `comm h` This limit is set at compile time before archiving `comm obj` into `ease h`

A.10 Interface with External Computer System

The prototype version of *Ease* availed of a dual port memory resource for data exchange between the target DSP system and an external computer system The communication process is governed by the application specific *Ease* include file `Dsp_if h` The file `Dsp_if h` contains symbolic constants which reference specific locations in absolute memory within the dual port memory space and declares prototypes for functions which application tasks can call to read and write from these locations

The functions following are the *Ease* functions declared in `Dsp_if h` for use by application tasks

- `EaseDspWordOut( )`
- `EaseDspWordIn( )`
- `EaseDspFloatOut( ),`
- `EaseGetDspPtr( )`

The prototype version of *Ease* employed an IBM PC for on-line user interface with a running *Ease* application The PC could access the DSP dual port memory via ports in the PC I/O memory space Prototype *Ease* applications employed an executable C file to conduct user interface from the PC side This C file is compiled including the *Ease* include file `Dsp_if h` This ensures that an application and the user interface functions agree on memory locations for data exchange The file `Dsp_if h` also contains port address information for use by the PC resident C file

The skeleton file `ui_skel c` contains examples of how to use the functions to read and write to the DSP dual port memory space from the PC

A.10.1 EaseDspWordOut( )

Prototype

```
int EaseDspWordOut(int dest,int word),
```

Arguments

- `int dest` This indicates to *Ease* where in absolute memory to place the output 32-bit `word` so that it can be accessed by an external computer system via a dual port memory interface
- `int word` This is the 32-bit word to be transfered by *Ease* to an external computer system via a dual port memory interface

Return Value

- `int` The non zero return values following are symbolic constants denoting errors and are defined in the application specific *Ease* include file `dsp_if h`
  - A zero is returned if operation is successful
  - `ADD_OUT_OF_RANGE` is returned if `dest` indicates an address which is outside the dual port memory area

Description This call is used by an application task to place 32-bit word in absolute memory with a view to interfacing with an external computer system via dual port memory space

This call places the word at a specific absolute memory location in the dual port memory which typically would be referenced through a symbolic constant defined in the *Ease* application specific include file `dsp_if h`

Constraints

- `int dest` This must be a positive integer not greater than the physical size of the dual port memory space (specified by the symbolic constant `DUAL` defined in `tms_if h`)

A.10.2 EaseDspWordIn( )

Prototype

```
int EaseDspWordIn(int source),
```

Arguments

- `int Source` This indicates to *Ease* a specific location in absolute memory which an external computer system has access to via dual port memory space

Return Value

- `int` This is the 32-bit word at the memory location that `source` indicates

Description This call is used by an application task to read the 32-bit word residing at a specific location in absolute memory with a view to interfacing with an external computer system via dual port memory space The absolute address is referenced by `source` displaced by the location of the dual port memory

The actual memory location of the input word would typically be referenced through a symbolic constant defined in the application specific *Ease* include file `dsp_if h`

Constraints

- `int dest` This is a positive integer not greater than the physical size of the dual port memory space (specified by the symbolic constant `DUAL` defined in `tms_if h`)

A.10.3 EaseDspFloatOut( )

Prototype

```
int  EaseDspFloatOut(int dest,float word),
```

Arguments

- `int dest` This indicates to *Ease* where in absolute memory to place the output floating point value so that it can be accessed by an external computer system via a dual port memory interface
- `float word` This is the floating point value to be transfered by *Ease* to an external computer system via a dual port memory interface

Return Value

- `int` The non zero return values following are symbolic constants denoting errors and are defined in the application specific *Ease* include file `dsp_if h`
  - A zero is returned if operation is successful
  - `ADD_OUT_OF_RANGE` is returned if `dest` indicates an address which is outside the dual port memory area

Description This call is used by an application task to place a floating point value in absolute memory with a view to interfacing with an external computer system via dual port memory space

This call places the floating point `word` at a specific absolute memory location in the dual port memory which typically would be referenced through a symbolic constant defined in the *Ease* application specific include file `dsp_if h`

Constraints

- `int dest` This must be a positive integer not greater than the physical size of the dual port memory space This size is specified by the symbolic constant `DUAL` defined in `tms_if h`

A.10.4 EaseGetDspPtr( )

Prototype

```
void *  EaseGetDspPtr(int memref),
```

Arguments

- `int memref` This indicates to *Ease* a specific location in absolute memory which an external computer system has access to via dual port memory space

Return Value

- `void *` Any non NULL return value returned is a pointer to the memory location indicated by `memref`

Description This call is used by an application task to obtain a pointer to a specific location in absolute memory with a view to interfacing with an external computer system via dual port memory space The absolute address is referenced by `memref` displaced by the location of the dual port memory

The `memref` argument would typically reference the dual port memory space through a symbolic constant defined in the application specific *Ease* include file `dsp_if h`

Constraints

- `int memref` This must be a positive integer not greater than the physical size of the dual port memory space If `memref` is invalid then the function will return a NULL pointer

A.11 Prototypes of UI-LIB

A.11.1 readword( )

Prototype

```
long   readword(long source),
```

Arguments

- `long source` This is the absolute address of a location in the dual port memory

Return Value

- `long int` This is a 32-bit word at location `source`

Description This call is used to fetch a 32-bit word from an absolute address in the dual port memory for PC consumption

Constraints

- `long source` This must be a positive integer not greater than the physical size of the dual port memory space (specified by the symbolic constant `DUAL` defined in `tms_if h`)

A.11.2 writeword( )

Prototype

```
void   writeword(long dest,long data),
```

Arguments

- `long dest` This is the absolute address of a location in the dual port memory
- `long data` This is a 32-bit word to be written to the an absolute address of a location in the dual port memory

Return Value

- `void`

Description This call is used to place a 32-bit word at an absolute address in the dual port memory for DSP consumption

Constraints

- `long source` This must be a positive integer not greater than the physical size of the dual port memory space(specified by the symbolic constant `DUAL` defined in `tms_if h`)

A.11.3 tmstoIEEE( )

Prototype

```
double tmsftoibmf(long a),
```

Arguments

- `long a` This is the TMS320C30 32-bit floating point representation to be converted

Return Value

- `double` This is an IEEE format double precision float

Description This call is used convert a 32-bit floating point format word previously obtained from the DSP and convert it to the IEEE format

A.12 Installing *Ease* in an IBM PC

The prototype version of *Ease* is developed for a TMS320C30 platform Programs written under *Ease* use the Texas instruments microprocessor development tools The particular hardware environment is the TMS320C30 board developed by Loughborough Sound Images *Ease* is completely compatible with this hardware platform

A.12.1 Obtaining *Ease*

The release version of *Ease* can be obtained through access to an anonymous `ftp` site located in Dublin City University named `ftp eeng dcu ie` Upon successful connection to the ftp site give the user name as `ftp` to log in as a guest and the password as your Internet email address

Ease is to be found in the `pub/power/ease` directory Declare the transfer protocol as binary mode by using the `bin` command and transfer the files `README TXT` and `EASE ZIP` to your PC using the `get` command

Use `pkunzip` with the switch `-d` to extract the files and subdirectories of *Ease* The dos command line would look like this

```
pkunzip -d ease zip
```

The file README TXT gives a detailed description of how to install *Ease*

A.12.2 Setting up *Ease*

The LSI systems board is connected to the PC via the PC I/O ports The base address of this port must be written to the data file port dat in the

```
\ease\bin
```

directory The port address is to be given in ASCII hexadecimal format In the prototype version of *Ease* port dat contains the address 290

The TMS320C30 C compiler uses the environment variable C_DIR to specify alternate directories that contain #include files The path of the *Ease* include files must be appended to the existing C_DIR paths The Appended entry would look like the following if *Ease* is installed on the C drive

```
C_DIR=C \ease\include
```

A.12.3 Running an *Ease* Application

An application loaded into the c30 memory and executed through the program 30run exe 30run operates on out files To run an application type 30run and the name of the out file

To interface with the PC the *Ease* application requires a program to be run from the PC The file ui-min gives the minimum interface with *Ease* Each application will have its own user interface program associated with it

A.12.4 Platform Specific Considerations

The prototype version of *Ease* was developed on a TMS320C30 system board developed by Loughborough Sound Images ltd If *Ease* is to be used on another hardware platform then alterations would have to be made to certain *Ease* modules

A.12.4.1 Configuration for a different TMS320C30 System

If *Ease* is to be utilised on a hardware platform which is different from the prototypes but one that still uses the TMS320C30 microprocessor system then the majority of the code of *Ease* does not need to be changed The only changes required are in the interface module `tms_if asm` and in the TMS linker command file `ease cmd`

`Tms_if` needs to be altered as it references absolute memory and as it relies on there being a dual port memory resource

`Ease cmd` describes the hardware configuration to the TMS linker so it can generate `out` files from object files

A.12.4.2 Configuration for a Different Microprocessor System

If the target platform's hardware system is based on microprocessor different from the prototypes, all the *Ease* C source files are still valid All of the assembly modules would have to be altered for the specific processor

Appendix B

Code Listings

B.1 *Ease* Source Code Listings

B.1.1 Kernel.h

```
/*      *

FILE  Kernel h

This file contains
a) Module to initialize Task control blocks
b) Scheduler for servicing clock ticks
c) General scheduler called after interrupts
d) System timer services

David Doyle 6/10/94

Date        initials        history
************************************************************
2/7/93      D D             PRE-RELEASE

*/

#define NULL (0)
#define TRUE 1
#define FALSE 0
#define ROGUE -1
#define SYSTEM_TIMER_MSG 4
#define INCORRECT_TIMER_MODE -3
#define RESET_WHILE_ACTIVE   -4


#define MONOSTABLE 0
#define ASTABLE 1
```

```
#define public
#define private static

#define EaseLock     1
#define EaseUnlock   0

#define MAX_TASKS 15
#define PRIORITY_LEVELS 8
#define QUANTUM 2

#define TIMER0_INTVEC     0x9

typedef void (*EaseTaskId_t)(void),

typedef struct EaseTaskCtrl_s
{
  int       blocked_status,        /* FALSE if not blocked else TRUE    */
  int       quantum_tick,
  int       task_sp,
  int       task_id,               /* integer to identify task          */
  EaseTaskId_t root,                  /* address of root function of task  */
  struct EaseTaskCtrl_s *next_member, /* pointer to next member            */
} EaseTaskCtrl ,


extern EaseTaskCtrl  EaseTask[MAX_TASKS+1],
extern EaseTaskCtrl* EaseTaskPtr[PRIORITY_LEVELS],
extern int           EaseCurrentTask,
extern int           EaseCurrentPriority,
extern int           EaseTaskPr[MAX_TASKS],
extern int           EaseNtasks[PRIORITY_LEVELS],
extern void          EaseScheduleAfterInt(void),
extern void          EaseForge(void),
extern void          EaseIdle(void),
extern void          EaseSystemTimerInit(int freq),
extern int           EaseSystemTimerSet(int ticks,int channel,int mode_c),
extern char*         EaseErrorMessage,
extern int EaseClockTick,
extern int EaseScheduleCount,

extern void EaseInt0Int(void),
```

B.1.2 Kernel.c

```
/*****************************************************/
/*  FILE  kernel c                                   */
/*****************************************************/

#ınclude<strıng h>
#ınclude<stdlıb h>
#ınclude"kernel h"
#ınclude"comm h"    /*   ıncluded for channel_ınıt(),   */

publıc  EaseTaskCtrl  EaseTask[MAX_TASKS+1],
publıc  EaseTaskCtrl* EaseTaskPtr[PRIORITY_LEVELS],
publıc  ınt EaseCurrentTask,
publıc  ınt EaseCurrentPrıorıty,
publıc  ınt EaseNtasks[PRIORITY_LEVELS]={ 0,0,0,0,0 },
publıc  ınt EaseScheduleCount=0,
publıc  ınt EaseTaskPr[MAX_TASKS],
publıc  ınt EaseSystemTımerActıve=FALSE,
publıc  ınt EaseSystemTımerNtıcks=0,

extern  voıd EaseTımer0Int(voıd),

prıvate ınt system_tımer_perıod_of_tıcks,
prıvate ınt system_tımer_mode,
prıvate ınt system_tımer_channel,
prıvate ınt ıdle_prıorıty=0,
prıvate ınt count=0,
prıvate ınt system_tımer_ınıtıalısed=FALSE,


voıd EaseCreate(EaseTaskId_t functıon,ınt prıorıty,ınt stack_alloc)
{
  ıf(ıdle_prıorıty < prıorıty)
     ıdle_prıorıty=prıorıty,
  EaseTask[count] blocked_status=FALSE,
  EaseTask[count] quantum_tıck=0,
  EaseStack(count,stack_alloc,functıon),
  EaseTask[count] task_ıd=count,
  EaseTask[count] root=functıon,
  EaseTaskPr[count]=prıorıty,
  ıf(count++>MAX_TASKS)
  {
    strcpy(EaseErrorMessage,"Too Many Tasks created ın EaseForge()"),
    EasePanıc(),
  }
}

voıd EaseGroup(voıd)
{
```

```
  EaseTaskCtrl * ptr,* first,
  int       i,j,warning=FALSE,

  for(j=0,j<PRIORITY_LEVELS,j++)
  {
    first=(NULL),
    for(i=0,i<(count),i++)
    {
      if(EaseTaskPr[i]>=PRIORITY_LEVELS)
      {
strcpy(EaseErrorMessage,
       "Number of task priority levels exceeds max in EaseForge"),
EasePanic(),
      }

      if(EaseTaskPr[i]==j)
      {
EaseNtasks[j]++,
if(EaseNtasks[j]==1)
{
  EaseTaskPtr[j] = &EaseTask[i],
  ptr          = &EaseTask[i],
  first        = &EaseTask[i],
}
else
{
  ptr->next_member = &EaseTask[i],
  ptr = &EaseTask[i],
}
      }
    }
    if(first==(NULL))
      warning=TRUE,
    if(warning==TRUE && first!=(NULL))
    {
      strcpy(EaseErrorMessage,"Priority Rules not Respected in EaseForge"),
      EasePanic(),
    }
    ptr->next_member=first,
  }
}


void EaseInit(void)
{
  EaseForge(),
  if(system_timer_initialised==FALSE)
  {
    strcpy(EaseErrorMessage,"Ease System timer not initialised"),
    EasePanic(),
```

```
  }
  EaseCreate(EaseIdle,(idle_priority+1),0x50),
  EaseGroup(),
  EaseChannelInit(),
  EaseCurrentTask=EaseTaskPtr[0]->task_id,
  EaseCurrentPriority=0,
}

void EaseScanLevelAfterInt(int *z)
{
  EaseTaskCtrl *sweeper,

  sweeper=EaseTaskPtr[*z]->next_member,

  while(sweeper!=EaseTaskPtr[*z])
  {
    if(sweeper->blocked_status==FALSE)
    {
      EaseCurrentTask=sweeper->task_id,
      EaseCurrentPriority=*z,
      EaseTaskPtr[*z]=sweeper,
      EaseRun(EaseCurrentTask),
    }
    sweeper=sweeper->next_member,
  }
}

void EaseScheduleAfterInt(void)
{
  int z,
  z=0,
  EaseScheduleCount++,
  while(TRUE)
  {
    if(EaseTaskPtr[z]->blocked_status==FALSE)
    {
      EaseCurrentPriority=z,
      EaseCurrentTask=EaseTaskPtr[z]->task_id,
      EaseRun(EaseCurrentTask),
    }
    else
    {
      EaseScanLevelAfterInt(&z),
    }
    z++,
  }
}

int EaseSystemTimerSet(int ticks,int channel,int mode_c)
{
```

```
  int status=0,
  asm("  TRAP 0"),
  if(0>channel||channel>CHANNELS)
  {
    asm("  TRAP 1"),
    return(NOT_VALID_CHANNEL),
  }
  if(mode_c!=MONOSTABLE&&mode_c!=ASTABLE)
  {
    asm("  TRAP 1"),
    return(INCORRECT_TIMER_MODE),
  }
  if(EaseSystemTimerNticks!=0)
     status=RESET_WHILE_ACTIVE,
  EaseSystemTimerNticks=ticks,
  system_timer_period_of_ticks=ticks,
  system_timer_mode=mode_c,
  EaseSystemTimerActive=TRUE,
  system_timer_channel=channel,
  asm("  TRAP 1"),
  return(status),
}


void EaseSystemTimerMsg(void)
{
  EaseTaskCtrl *sweeper,
  int i,message[2],size,tail,head,lucky_task,
  EaseChanCtrl *ch,

  ch=&EaseChannel[system_timer_channel],
  tail=ch->tail_q,
  head=ch->head_q,
  size=2,

  message[0]=SYSTEM_TIMER_MSG,,
  message[1]=EaseClockTick,
  if(system_timer_mode==ASTABLE)
  {
    EaseSystemTimerNticks=system_timer_period_of_ticks,
  }
  else
  {
    EaseSystemTimerActive=FALSE,
  }

  if((ch->source_flag==FALSE)&&(tail!=head))
  {
    EaseTransfer((int *)message,(int *)ch->msg_q[tail],size),
```

```
    lucky_task=ch->id_q[tail],
    EaseTask[lucky_task] blocked_status=FALSE,
    EaseRendezvousRedemer[lucky_task]=(EaseTaskId_t)(NULL),
    EaseSenderMsgSize[lucky_task]=size,
    if(++ch->tail_q==MAX_MESSAGES)
       ch->tail_q=0,
    EaseScheduleAfterInt(),
  }
}

void EaseScanLevel(void)
{
  EaseTaskCtrl *sweeper,

  if(EaseSystemTimerActive==TRUE&&EaseSystemTimerNticks==0)
    EaseSystemTimerMsg(),

  sweeper=EaseTask[EaseCurrentTask] next_member,

  while(TRUE)
  {
    if(sweeper->blocked_status==0)
    {
      EaseCurrentTask=sweeper->task_id,
      EaseTaskPtr[EaseCurrentPriority]=sweeper,
      EaseRun(EaseCurrentTask),
    }
    sweeper=sweeper->next_member,
  }
}

void EaseIdle(void)
{
  while(TRUE),
}

void EaseSystemTimerInit(int freq)
{
  float tick_period,
  system_timer_initialised=TRUE,
  if(EaseSetVec(TIMER0_INTVEC,EaseTimer0Int)!=0)
     EasePanic(),
  tick_period=1/(float)freq,
  EaseTimer0( (int)(tick_period*1/120E-9)),
}

void main(void)
{
  strcpy(EaseErrorMessage,""),
  EaseInit(),
```

```
  if(EaseSetIEreg(TIMER0_INTVEC)!=0)
     EasePanic(),
  EaseRun(EaseCurrentTask),
  while(TRUE),
}
```

B.1.3 Comm.h

```
/*

FILE  comm h

This file contains ease communication modules

David Doyle 10/9/94

Date        initials        history
************************************************************
10/9/94      D D              PRE-RELEASE

*/

#define TRUE 1
#define FALSE 0

#define CHANNELS 8
#define MAX_MESSAGES 5
#define SIGNAL_CH 5

#define NOT_VALID_CHANNEL -1
#define MSG_TOO_LARGE_FOR_RECEIVER -2

typedef struct EaseChanCtrl_s
{
  int source_flag,
  int id_q[MAX_MESSAGES],
  int *msg_q[MAX_MESSAGES],
  int size_q[MAX_MESSAGES],
  int head_q,
  int tail_q,
} EaseChanCtrl,

extern EaseChanCtrl EaseChannel[CHANNELS],
extern void EaseChannelInit(void),
extern EaseTaskId_t EaseRendezvousRedemer[MAX_TASKS],
extern int EaseSenderMsgSize[MAX_TASKS],
extern int EaseSendCount,
extern int EaseReceiveCount,

extern int EaseReceive(int src_ch,void  msg[],
       int max_msg_size,
       int *msg_size,
       EaseTaskId_t* rendezvous_tsk),
extern int EaseSend(int dst_ch,void msg[],
    int msg_size,
    EaseTaskId_t* rendezvous_tsk),
```

B.1.4 Comm.c

```
/*****************************************************
* File comm c handles intertask communication        *
*****************************************************/

#include "kernel h"
#include "comm h"

extern void EaseSchedule(void),
extern void EaseTransfer(int * source,int * dest ,int size),

public int EaseSendCount=0,
public int EaseReceiveCount=0,

public EaseChanCtrl EaseChannel[CHANNELS],
public EaseTaskId_t EaseRendezvousRedemer[MAX_TASKS],
public int EaseSenderMsgSize[MAX_TASKS],

void EaseChannelInit(void)
{
  int i,

  for(i=0,i<CHANNELS,i++)
  {
    EaseChannel[i] source_flag=FALSE,
    EaseChannel[i] tail_q=0,
    EaseChannel[i] head_q=0,
  }
}

int EaseSend(int dst_ch,void msg[],
     int msg_size,
     EaseTaskId_t* rendezvous_tsk)
{
  int i=0,head,tail,lucky_task,
  EaseChanCtrl *ch,

  asm("  TRAP 0"),
  EaseSendCount++,

  if(0>dst_ch||dst_ch>CHANNELS)
     return(NOT_VALID_CHANNEL),
  ch=&EaseChannel[dst_ch],
  head=ch->head_q,
  tail=ch->tail_q,

  if((tail==head) || (ch->source_flag==TRUE))
  {
    ch->source_flag=TRUE,
```

```
    ch->id_q[head]=EaseTaskPtr[EaseCurrentPriority]->task_id,
    ch->size_q[head]=msg_size,
    ch->msg_q[head]=(int *)msg,
    EaseTaskPtr[EaseCurrentPriority]->blocked_status=TRUE,
    if(++ch->head_q==MAX_MESSAGES)
      ch->head_q=0,

    EaseSchedule(),
    *rendezvous_tsk=EaseRendezvousRedemer[EaseCurrentTask],
    return(0),
  }
  else
  {
    lucky_task=ch->id_q[tail],
    if(msg_size > ch->size_q[tail])
      return(MSG_TOO_LARGE_FOR_RECEIVER),
    EaseTransfer((int *)msg,(int *)ch->msg_q[tail],msg_size),
    EaseTask[lucky_task] blocked_status=FALSE,

    *rendezvous_tsk=EaseTask[lucky_task] root,
    EaseRendezvousRedemer[lucky_task]=EaseTask[EaseCurrentTask] root,
    EaseSenderMsgSize[lucky_task]=msg_size,

    if(++ch->tail_q==MAX_MESSAGES)
      ch->tail_q=0,
    if(EaseTaskPr[lucky_task]>EaseCurrentPriority)
    {
      asm("  TRAP 1"),
      return(0),
    }
    else
    {
      EaseSchedule(),
      return(0),
    }
  }
}

int EaseReceive(int src_ch,
void msg[],
int max_msg_size,
int *msg_size,
EaseTaskId_t* rendezvous_tsk)
{
  int i=0,head,tail,lucky_task,
  EaseChanCtrl *ch,

  asm("  TRAP 0"),
  EaseReceiveCount++,
```

```
if(0>src_ch||src_ch>CHANNELS)
  return(NOT_VALID_CHANNEL),
ch=&EaseChannel[src_ch],
head=ch->head_q,
tail=ch->tail_q,

if((tail==head) || (ch->source_flag==FALSE))
{
  ch->source_flag=FALSE,
  ch->id_q[head]=EaseTaskPtr[EaseCurrentPriority]->task_id,
  ch->size_q[head]=max_msg_size,
  ch->msg_q[head]=(int *)msg,
  EaseTaskPtr[EaseCurrentPriority]->blocked_status=TRUE,
  if(++ch->head_q==MAX_MESSAGES)
    ch->head_q=0,

  EaseSchedule(),
  *rendezvous_tsk=EaseRendezvousRedemer[EaseCurrentTask],
  *msg_size=EaseSenderMsgSize[EaseCurrentTask],
  return(0),
}
else
{
  lucky_task=ch->id_q[tail],
  if(max_msg_size < ch->size_q[tail])
    return(MSG_TOO_LARGE_FOR_RECEIVER),
  EaseTransfer((int *)ch->msg_q[tail],
(int *)msg,
ch->size_q[tail]),
  *msg_size=ch->size_q[tail],
  EaseTask[ lucky_task ] blocked_status=FALSE,

  *rendezvous_tsk=EaseTask[lucky_task] root,
  EaseRendezvousRedemer[lucky_task]
    =EaseTask[EaseCurrentTask] root,
  EaseSenderMsgSize[lucky_task]=*msg_size,

  if(++ch->tail_q==MAX_MESSAGES)
    ch->tail_q=0,

  if(EaseTaskPr[lucky_task]>EaseCurrentPriority)
  {
    asm("  TRAP 1"),
    return(0),
  }
  else
  {
    EaseSchedule(),
    return(0),
  }
```

```
    }
}
```

B.1.5 Tim.h

```
/*

FILE  TIM h

This module handles application timer services

David Doyle 7/7/93

Date        initials        history
************************************************************
7/7/93      D D             PRE-RELEASE

*/

#define TRUE  1
#define FALSE 0

#define TIMER1_INTVEC 0xA
#define CONVERSION_COMPLETE_INTVEC 0x2
#define END_OF_CONVERSION_INT 2
#define APPLICATION_TIMER_MSG 3


extern void EaseApplicationTimerInit(int freq),
extern int EaseApplicationTimerSet(int ticks,int channel,int mode),
extern int EaseSamplerSet(int channel),
extern int EaseInt1Count,
extern int EaseLostInt1,
```

B.1.6 Tim.c

```
/*****************************************************************/
/* FILE  tim2 c                                                  */
/*****************************************************************/

#include"kernel h"
#include"comm h"
#include"tim h"
#include<stdlib h>

extern void EaseTimer1Int(void),
extern void EaseInt1Int(void),

public int EaseInt1Count,
public int EaseLostInt1=0,
public void EaseTimer1Handler(void),

private int tim_channel,
private int insignia,
private int n_ticks=0,
private int period_of_ticks,
private int mode,

void EaseApplicationTimerInit(int freq)
{
  float period,
  int   t_reg,

  period=(float)1/freq,
  t_reg=(int)(period/(120E-9)),
  EaseSetVec(TIMER1_INTVEC,EaseTimer1Int),
  EaseSetVec(CONVERSION_COMPLETE_INTVEC,EaseInt1Int),
  EaseTimer1(t_reg),
  EaseInt1Count=0,
}

int EaseApplicationTimerSet(int ticks,int channel,int mode_c)
{
  int status=0,
  asm("  TRAP 0"),
  if(0>channel||channel>CHANNELS)
  {
    asm("  TRAP 1"),
    return(NOT_VALID_CHANNEL),
  }
  if(mode_c!=MONOSTABLE&&mode_c!=ASTABLE)
  {
    asm("  TRAP 1"),
    return(INCORRECT_TIMER_MODE),
```

```
  }
  if(n_ticks!=0)
    status=RESET_WHILE_ACTIVE,
  n_ticks=ticks,
  period_of_ticks=ticks,
  mode=mode_c,
  insignia=APPLICATION_TIMER_MSG,
  tim_channel=channel,
  EaseSetIEreg(TIMER1_INTVEC),
  asm("  TRAP 1"),
  return(status),
}

int EaseSamplerSet(int channel)
{
  int status=0,ticks=1,mode_c=ASTABLE,
  asm("  TRAP 0"),
  if(0>channel||channel>CHANNELS)
  {
    asm("  TRAP 1"),
    return(NOT_VALID_CHANNEL),
  }
  if(n_ticks!=0)
    status=RESET_WHILE_ACTIVE,
  n_ticks=ticks,
  period_of_ticks=ticks,
  mode=mode_c,
  insignia=END_OF_CONVERSION_INT,
  tim_channel=channel,
  EaseSetIEreg(CONVERSION_COMPLETE_INTVEC),
  asm("  TRAP 1"),
  return(status),
}

void EaseTimer1IntHandler(void)
{
  int i,message[2],size,tail,head,lucky_task,
  EaseChanCtrl *ch,

  ch=&EaseChannel[tim_channel],
  tail=ch->tail_q,
  head=ch->head_q,

  size=2,

  message[0]=insignia,
  message[1]=EaseInt1Count++,

  if(--n_ticks==0)
  {
```

```
    if(mode==ASTABLE)
    {
      n_ticks=period_of_ticks,
    }
    else
    {
      EaseUnsetIEreg(CONVERSION_COMPLETE_INTVEC),
      EaseUnsetIEreg(TIMER1_INTVEC),
    }
    if((ch->source_flag==FALSE)&&(tail'=head))
    {
      EaseTransfer((int *)message,(int *)ch->msg_q[tail],size),

      /* Unblock receiver         */
      lucky_task=ch->id_q[tail],
      EaseTask[lucky_task] blocked_status=FALSE,
      EaseRendezvousRedemer[lucky_task]=(EaseTaskId_t)(NULL),
      EaseSenderMsgSize[lucky_task]=size,

      if(++ch->tail_q==MAX_MESSAGES)
ch->tail_q=0,

      EaseScheduleAfterInt(),
    }
    else
    {
      EaseLostInt1++,
    }
  }
  EaseRun(EaseTaskPtr[EaseCurrentPriority]->task_id),
}
```

B.1.7 Int0.h

```
/*

FILE  INT0 h

This module deals with interrupts on the Int0 line

David Doyle 2/9/93

Date        initials        history
************************************************************
2/9/93      D D             PRE-RELEASE

*/

#define INT0_VEC 0x1
#define INT0_INT 1

extern int EaseInt0Init(int channel),
extern int EaseInt0Count,
extern int EaseLostInt0,
```

B.1.8 Int0.c

```
/*****************************************************************/
/* FILE  int0 c                                                  */
/*****************************************************************/

#include"kernel h"
#include"comm h"
#include"int0 h"

public int EaseInt0Count,
public int EaseLostInt0=0,
public void EaseInt0Handler(void),
private int int0_channel,

int EaseInt0Init(int channel)
{
  int0_channel=channel,
  EaseSetVec(INT0_VEC,EaseInt0Int),
  EaseInt0Count=0,
  EaseSetIEreg(INT0_VEC),
  return(0),
}

void EaseInt0Handler(void)
{
  int i,message[2],size,tail,head,lucky_task,
  EaseChanCtrl *ch,

  ch=&EaseChannel[int0_channel],
  tail=ch->tail_q,
  head=ch->head_q,

  size=2,

  message[0]=INT0_INT,
  message[1]=EaseInt0Count++,

  if((ch->source_flag==FALSE)&&(tail'=head))
  {
    EaseTransfer((int *)message,(int *)ch->msg_q[tail],size),

    /* Unblock receiver         */
    lucky_task=ch->id_q[tail],
    EaseTask[lucky_task] blocked_status=FALSE,
    EaseRendezvousRedemer[lucky_task]=(EaseTaskId_t)(NULL),
    EaseSenderMsgSize[lucky_task]=size,

    if(++ch->tail_q==MAX_MESSAGES)
      ch->tail_q=0,
```

```
    EaseScheduleAfterInt(),
  }
  else
  {
    EaseLostInt0++,
    EaseRun(EaseTaskPtr[EaseCurrentPriority]->task_id),
  }
}
```

B.1.9 Kextra.h

```
/*

FILE  Kextra h

This file contains the prototypes to the functions
contained in kextra asm



David Doyle 6/10/94

Date        initials        history
************************************************************
2/7/93      D D             PRE-RELEASE

*/

extern int EaseSetVec(int intVector, void (*theFunction)(void) ),
extern int EaseSetIEReg(int theBit),
extern int EaseUnsetIEreg(int theBit),
extern void EaseTimer0(int theOffset),
extern void EaseTimer1(int theOffset),
extern void EaseStack(int TaskID,
      int theStackAlloc,
      void(*theFunction)(void)),
extern int * EaseGetSP(void),
extern void EasePanic(void),
extern void EaseRun(int theTaskID),
extern void EaseTimer0Int(void),
extern void EaseTimer1Int(void),
extern void EaseSchedule(void),
extern void EaseInt0Int(void),
extern void EaseInt1Int(void),
extern void EaseDisableInterrupts(void),
extern void EaseTransfer(int * theSrc,int * theDest,int theSize),
extern void EaseRootExit(void),
```

B.1.10 Kextra.asm

```
*************************************************************************
*
* KEXTRA asm David Doyle School of Electronic Engineering DCU
* DATE 9/6/94
*
* Contains
*   1) _EaseSetVec        , Sets interrupt vectors
*   2) _EaseSetIEreg     , Sets Interrupt enable reg
*   2a) _EaseUnsetIEreg   ,
*   3) _EaseTimer0          , initializes clock
*   4) _EaseTimer1          , initializes timer1
*   5) _EaseStack           , Puts initial context of task on task's stack
*   6) _EaseGetSp          , snatches SP
*   7) _EasePanic
*   8) _EaseRun             , Pops context off new tasks stack
*   9) _EaseTimer0Int      , Context switching rountines that call
*  10) _EaseTimer1Int     , relevent handlers
*  11) _EaseSchedule
*  12) _EaseInt0Int
*  13) _EaseInt1Int
*  14) _EaseDisableInterrupts , Safely changes ST 'GIE ' bit
*  15) _EaseTransfer
*  16) _EaseRootExit
*
*************************************************************************

, initalize vectors to reset
   sect    int01
   word   _c_int00
   sect    int02
   word   _c_int00
   sect    int03
   word   _c_int00
   sect    int04
   word   _c_int00
   sect    int05
   word   _c_int00
   sect    int06
   word   _c_int00
   sect    int07
   word   _c_int00
   sect    int08
   word   _c_int00
   sect    int09
   word   _c_int00
   sect    int10
   word   _c_int00
   sect    int11
```

```
   sect    trap0
   word   _trap0
   sect    trap1
   word   _trap1

FP                  set    AR3
QUANTUM             set    2
ALLOC               set    400h
SIZE_TASK_STRUCTURE set    6
TIMER_RESET         set    601h
TIMER_GO            set    6c1h
DUAL                set    30000h

   sect    " cinit"
   word    1,_EaseErrorMessage
   word    DUAL
   globl   _EaseErrorMessage
   bss     _EaseErrorMessage,1


, base task stack section defined
 sect   " t_stack"
B_SP     word   0

 data

TIMER_CTRL_0        word   808020h
PERIOD_REG_0        word   808028h
TIMER_CTRL_1        word   808030h
PERIOD_REG_1        word   808038h

ADD_TIMER1_INT      word   _EaseTimer1Int
ADD_INT0_INT        word   _EaseInt0Int
EasePanicAddress    word   _EasePanic

BASE_SP             word   B_SP

BASE_TASK_TABLE     word   _EaseTask
TASK_PTR            word   _EaseTaskPtr
N_TASKS             word   _EaseNtasks

EaseRootExitAddress word   _EaseRootExit
RootExitStringAdd   word   EaseRootExitString
EaseRootExitString  byte   " Illegal Exit form root task "
     word   00H , T H E    T E R M I N A T O R

, symbols used

  text
```

```
        global _trap0
        global _trap1
        global _EaseSetVec
        global _EaseSetIEreg
        global _EaseUnsetIEreg
        global _EaseTimer0
        global _EaseTimer1
        global _EaseStack
        global _EaseGetSp
        global _EaseRun
        global _EasePanic
        global _EaseTimer0Int
        global _EaseTimer1Int
        global _EaseSchedule
        global _EaseInt0Int
        global _EaseInt1Int
        global _EaseDisableInterrupts
        global _EaseTransfer
        global _EaseRootExit

        global _EaseInt0Handler

        global _EaseTimer1IntHandler
        global _EaseScheduleAfterInt
        global _EaseScanLevel

        global _c_int09
        global _c_int00

        global _EaseTaskPtr
        global _EaseCurrentTask
        global _EaseCurrentPriority
        global _EaseSystemTimerActive
        global _EaseSystemTimerNticks
        global _EaseSystemTimerMsg

        global  _EaseTask
        global  _EaseNtasks
        global  _EaseChannel

        sect    " cinit"
        word    1,_EaseClockTick
        word    0
        globl   _EaseClockTick
        bss     _EaseClockTick,1

        text
_trap0
```

```
    RETS

_trap1
    RETI

_EaseSetVec
    PUSH        FP
    LDI    SP,FP
    LDI    *-FP(2),AR0
    LDI    *-AR3(3),R0
    LDI   AR0,R1
    SUBI  10,R1
    BGT   S1
    STI    R0,*AR0
    LDI   0,R0
    B     S2
S1
    LDI   -1,R0
S2
    LDI    *-FP(1),R1
    BD    R1
    LDI    *FP,FP
    NOP
    SUBI  2,SP

_EaseSetIEreg
    PUSH        FP
    LDI    SP,FP
    LDI   *-FP(2),R1
    CMPI  10,R1
    LDIGT 0,R0
    BGT   S3
    LDI    1,R0
    LSH   R1,R0
    ROR   R0
    OR    R0,IE
    LDI   0,R0
S3
    LDI    *-FP(1),R1
    BD    R1
    LDI    *FP,FP
    NOP
    SUBI  2,SP

_EaseUnsetIEreg
    PUSH        FP
    LDI    SP,FP
    LDI   *-FP(2),R1
    CMPI  10,R1
    LDIGT 0,R0
```

```
    BGT   S4
    LDI    1,R0
    LSH   R1,R0
    ROR   R0
    NOT   R0
    AND   R0,IE
    LDI   0,R0
S4
    LDI    *-FP(1),R1
    BD    R1
    LDI    *FP,FP
    NOP
    SUBI  2,SP

_EaseTimer0
    PUSH  FP
    LDI   SP,FP

    LDI   @TIMER_CTRL_0,AR0
    LDI   TIMER_RESET,R0
    STI   R0,*AR0
    LDI   @PERIOD_REG_0,AR1
    LDI   *-FP(2),R0       , Load argument count
    STI   R0,*AR1
    LDI   TIMER_GO,R0
    STI   R0,*AR0

    LDI    *-FP(1),R1
    BD    R1
    LDI    *FP,FP
    NOP
    SUBI  2,SP

_EaseTimer1
    PUSH  FP
    LDI   SP,FP

    LDI   @TIMER_CTRL_1,AR0
    LDI   TIMER_RESET,R0
    STI   R0,*AR0
    LDI   @PERIOD_REG_1,AR1
    LDI   *-FP(2),R0       , Load argument count
    STI   R0,*AR1
    LDI   TIMER_GO,R0
    STI   R0,*AR0

   LDI    *-FP(1),R1
    BD    R1
    LDI    *FP,FP
    NOP
```

```
    SUBI        2,SP

_EaseStack
    PUSH    FP
    LDI     SP,FP

    LDI     *-FP(2),AR0     , = count
    LDI     *-FP(3),AR1     , = stack_alloc
    LDI     *-FP(4),AR2     , = *function
    LDI     @BASE_SP,R1     , = Absolute address of task's stack

    LDI     SP,R0           , tempory storage for SP & FP
    LDI     R1,FP           , ST changed but OK at this stage
    LDI     R1,SP

    LDI     @EaseRootExitAddress,R2 , Panic if return from root task
    STI     R2,*FP

    PUSH    AR2             , Push inital context
    PUSH    ST
    PUSH    FP
    PUSH    R0
    PUSHF   R0
    PUSH    R1
    PUSHF   R1
    PUSH    R2
    PUSHF   R2
    PUSH    R3
    PUSHF   R3
    PUSH    R4
    PUSHF   R4
    PUSH    R5
    PUSHF   R5
    PUSH    R6
    PUSHF   R6
    PUSH    R7
    PUSHF   R7
    PUSH    AR0
    PUSH    AR1
    PUSH    AR2
    PUSH    AR4
    PUSH    AR5
    PUSH    AR6
    PUSH    AR7
    PUSH    IR0
    PUSH    IR1
    PUSH    BK
    PUSH    RC
    PUSH    RS
    PUSH    RE
```

```
    PUSH    DP

    MPYI    SIZE_TASK_STRUCTURE,AR0 , saves inital SP of task
    ADDI    @BASE_TASK_TABLE,AR0
    STI     SP,*+AR0(2)

    LDI     R0,SP           , restores system SP and FP
    LDI     R0,FP

    ADDI    AR1,R1          , allocate stack
    STI     R1,@BASE_SP

    LDI     *-FP(1),R1
    BD      R1
    LDI     *FP,FP
    NOP
    SUBI    2,SP


_EaseGetSp
  LDI     SP,R0
  SUBI    1,R0    , Take pc off stack
  RETS

_EasePanic

L1
  BR      L1


_EaseRun
  POP     AR0             , get task to run
  POP     AR0
  MPYI    SIZE_TASK_STRUCTURE,AR0
  ADDI    @BASE_TASK_TABLE,AR0
  LDI     *+AR0(2),SP     , change to tasks stack
  POP     DP
  POP     RE
  POP     RS
  POP     RC
  POP     BK
  POP     IR1
  POP     IR0
  POP     AR7
  POP     AR6
  POP     AR5
  POP     AR4
  POP     AR2
  POP     AR1
  POP     AR0
```

```
  POPF    R7
  POP     R7
  POPF    R6
  POP     R6
  POPF    R5
  POP     R5
  POPF    R4
  POP     R4
  POPF    R3
  POP     R3
  POPF    R2
  POP     R2
  POPF    R1
  POP     R1
  POPF    R0
  POP     R0
  POP     FP
  POP     ST
  RETI

_EaseTimer0Int
  PUSH    ST
  PUSH    AR3
  PUSH    R0
  PUSHF   R0

  LDI     @_EaseSystemTimerActive,R0
  BZ      S5
  LDI     @_EaseSystemTimerNticks,R0
  SUBI    1,R0
  STI     R0,@_EaseSystemTimerNticks
  BNZ     S5

  LDI     @_EaseClockTick,AR3
  ADDI    1,AR3
  STI     AR3,@_EaseClockTick

  LDI     @_EaseCurrentPriority,AR3
  ADDI    @N_TASKS,AR3
  LDI     *AR3,R0
  CMPI    1,R0
  BZD     SWITCH

  LDI     @TASK_PTR,AR3
  ADDI    @_EaseCurrentPriority,AR3
  LDI     *AR3,AR3

  LDI     *+AR3(1),R0
  ADDI    1,R0
  CMPI    QUANTUM,R0
```

```
  LDIZ    0,R0
  STI     R0,*+AR3(1)
  B       SWITCH

S5
  LDI     @_EaseCurrentPriority,AR3
  ADDI    @N_TASKS,AR3
  LDI     *AR3,R0
  CMPI    1,R0
  BZD     END_EaseTimer0Int
  LDI     @_EaseClockTick,AR3
  ADDI    1,AR3
  STI     AR3,@_EaseClockTick

  LDI     @TASK_PTR,AR3
  ADDI    @_EaseCurrentPriority,AR3
  LDI     *AR3,AR3
  LDI     *+AR3(1),R0
  ADDI    1,R0
  CMPI    QUANTUM,R0
  LDIZ    0,R0
  STI     R0,*+AR3(1)
  BZ      SWITCH


END_EaseTimer0Int
  POPF    R0
  POP     R0
  POP     AR3

  POP     ST
  RETI

SWITCH
  PUSH    R1
  PUSHF   R1
  PUSH    R2
  PUSHF   R2
  PUSH    R3
  PUSHF   R3
  PUSH    R4
  PUSHF   R4
  PUSH    R5
  PUSHF   R5
  PUSH    R6
  PUSHF   R6
  PUSH    R7
  PUSHF   R7
  PUSH    AR0
  PUSH    AR1
```

```
  PUSH    AR2

  PUSH    AR4
  PUSH    AR5
  PUSH    AR6
  PUSH    AR7

  PUSH    IR0
  PUSH    IR1
  PUSH    BK
  PUSH    RC
  PUSH    RS
  PUSH    RE
  PUSH    DP

  STI     SP,*+AR3(2)

  CALL    _EaseScanLevel
  CALL    _EasePanic


_EaseTimer1Int
  PUSH    ST
  PUSH    FP
  PUSH    R0
  PUSHF   R0
  PUSH    R1
  PUSHF   R1
  PUSH    R2
  PUSHF   R2
  PUSH    R3
  PUSHF   R3
  PUSH    R4
  PUSHF   R4
  PUSH    R5
  PUSHF   R5
  PUSH    R6
  PUSHF   R6
  PUSH    R7
  PUSHF   R7
  PUSH    AR0
  PUSH    AR1
  PUSH    AR2

  PUSH    AR4
  PUSH    AR5
  PUSH    AR6
  PUSH    AR7

  PUSH    IR0
```

```
  PUSH    IR1
  PUSH    BK
  PUSH    RC
  PUSH    RS
  PUSH    RE
  PUSH    DP

  LDI     @TASK_PTR,AR0
  ADDI    @_EaseCurrentPriority,AR0
  LDI     *AR0,AR0
  STI     SP,*+AR0(2)

  CALL _EaseTimer1IntHandler
  CALL _EasePanic

_EaseSchedule
  PUSH    ST
  PUSH    FP
  PUSH    R0
  PUSHF   R0
  PUSH    R1
  PUSHF   R1
  PUSH    R2
  PUSHF   R2
  PUSH    R3
  PUSHF   R3
  PUSH    R4
  PUSHF   R4
  PUSH    R5
  PUSHF   R5
  PUSH    R6
  PUSHF   R6
  PUSH    R7
  PUSHF   R7
  PUSH    AR0
  PUSH    AR1
  PUSH    AR2

  PUSH    AR4
  PUSH    AR5
  PUSH    AR6
  PUSH    AR7

  PUSH    IR0
  PUSH    IR1
  PUSH    BK
  PUSH    RC
  PUSH    RS
  PUSH    RE
  PUSH    DP
```

```
  LDI     @TASK_PTR,AR0
  ADDI    @_EaseCurrentPriority,AR0
  LDI     *AR0,AR0
  STI     SP,*+AR0(2)

  CALL _EaseScheduleAfterInt

  CALL _EasePanic

_EaseInt0Int
  PUSH    ST
  PUSH    FP
  PUSH    R0
  PUSHF   R0
  PUSH    R1
  PUSHF   R1
  PUSH    R2
  PUSHF   R2
  PUSH    R3
  PUSHF   R3
  PUSH    R4
  PUSHF   R4
  PUSH    R5
  PUSHF   R5
  PUSH    R6
  PUSHF   R6
  PUSH    R7
  PUSHF   R7
  PUSH    AR0
  PUSH    AR1
  PUSH    AR2

  PUSH    AR4
  PUSH    AR5
  PUSH    AR6
  PUSH    AR7

  PUSH    IR0
  PUSH    IR1
  PUSH    BK
  PUSH    RC
  PUSH    RS
  PUSH    RE
  PUSH    DP

  LDI     @TASK_PTR,AR0
  ADDI    @_EaseCurrentPriority,AR0
  LDI     *AR0,AR0
  STI     SP,*+AR0(2)
```

```
  IACK    *AR0

  CALL _EaseInt0Handler

  CALL _EasePanic

_EaseInt1Int
  PUSH    ST
  PUSH    FP
  PUSH    R0
  PUSHF   R0
  PUSH    R1
  PUSHF   R1
  PUSH    R2
  PUSHF   R2
  PUSH    R3
  PUSHF   R3
  PUSH    R4
  PUSHF   R4
  PUSH    R5
  PUSHF   R5
  PUSH    R6
  PUSHF   R6
  PUSH    R7
  PUSHF   R7
  PUSH    AR0
  PUSH    AR1
  PUSH    AR2

  PUSH    AR4
  PUSH    AR5
  PUSH    AR6
  PUSH    AR7

  PUSH    IR0
  PUSH    IR1
  PUSH    BK
  PUSH    RC
  PUSH    RS
  PUSH    RE
  PUSH    DP

  LDI     @TASK_PTR,AR0
  ADDI    @_EaseCurrentPriority,AR0
  LDI     *AR0,AR0
  STI     SP,*+AR0(2)

  CALL _EaseTimer1IntHandler
  CALL _EasePanic
```

```
_EaseDisableInterrupts
  PUSH IE
  LDI  0,IE
  NOP
  NOP
  AND  0DFFFh,ST
  POP  IE
  RETS

_EaseTransfer
  PUSH FP
  LDI  SP,FP

  LDI  *-FP(2),AR0
  LDI  *-FP(3),AR1
  LDI  *-FP(4),RC
  SUBI 1,RC
  RPTB R_EaseTransfer
  LDI  *AR0++,R0
R_EaseTransfer
  STI  R0,*AR1++

  LDI  *-FP(1),R1
  BD   R1
  LDI  *FP,FP
  NOP
  SUBI 2,SP


_EaseRootExit
  PUSH  FP
  LDI   SP,FP

  LDI   @RootExitStringAdd,AR0
  LDI   @_EaseErrorMessage,AR1

_CopyErrorString
  LDI   *AR0++,R0
  BZ    _CopyTaskID
  STI   R0,*AR1++
  B     _CopyErrorString
_CopyTaskID
  LDI   @_EaseCurrentTask,R0
  ADDI  30h,R0
  ADDI  1,R0
  STI   R0,*AR1++
  LDI   0h,R0
  STI   R0,*AR1
  B     _EasePanic
```

```
end
```

B.1.11 Tms_if.asm

```
************************************************
*  TMS_IF asm
*
*   Interface routines for C30 to PC and Analog
* Interface devices
*
************************************************
*
* Routines included are as follows
*
*
************************************************

   data

ADCADR       word 000804000h
POS_LIMIT    word 000007FFFh
NEG_LIMIT    word 0FFFF8000h
DUAL         word 000030000h
SIZEOFDUAL   word 00000ffffh

   text
FP  set AR3
   global _read_adc
_read_adc
  PUSH FP
  LDI  SP,FP
  LDI  @ADCADR,AR0
  ADDI *-FP(2),AR0
  LDI  *AR0,R0
  ASH  -16,R0

  LDI  *-FP(1),R1
  BD   R1
  LDI  *FP,FP
  NOP
  SUBI 2,SP


   global _out_dac
_out_dac
  PUSH   FP
  LDI    SP,FP

  LDI    @ADCADR,AR0
  ADDI   *-FP(2),AR0
  LDI    *-FP(3),R0
  CMPI   @POS_LIMIT,R0
```

```
  LDIGT  @POS_LIMIT,R0
  CMPI   @NEG_LIMIT,R0
  LDILT  @NEG_LIMIT,R0
  ASH    16,R0
  STI    R0,*AR0

  LDI    *-FP(1),R1
  BD     R1
  LDI    *FP,FP
  NOP
  SUBI   2,SP

   globl _EaseDspWordOut
_EaseDspWordOut
  PUSH   FP
  LDI    SP,FP

  LDI    @DUAL,AR0
  LDI    *-FP(2),R0
  BN     WordAddOutOfRange
  CMPI   @SIZEOFDUAL,R0
  BGT    WordAddOutOfRange
  ADDI   R0,AR0
  LDI    *-FP(3),R0
  STI    R0,*AR0
  LDI    0,R0
  BR     _ENDEaseDspWordOut
WordAddOutOfRange
  LDI    -1,R0
_ENDEaseDspWordOut
  LDI    *-FP(1),R1
  BD     R1
  LDI    *FP,FP
  NOP
  SUBI   2,SP

   globl _EaseDspWordIn
_EaseDspWordIn
  PUSH   FP
  LDI    SP,FP

  LDI    @DUAL,AR0
  ADDI   *-FP(2),AR0
  LDI    *AR0,R0

  LDI    *-FP(1),R1
  BD     R1
  LDI    *FP,FP
  NOP
  SUBI   2,SP
```

```
   globl _EaseDspFloatOut
_EaseDspFloatOut
  PUSH   FP
  LDI    SP,FP

  LDI    @DUAL,AR0
  LDI   *-FP(2),R0
  BN     FloatAddOutOfRange
  CMPI   @SIZEOFDUAL,R0
  BGT    FloatAddOutOfRange
  ADDI   R0,AR0
  LDF    *-FP(3),R0
  STF    R0,*AR0
  LDI    0,R0
  BR     _ENDEaseDspFloatOut
FloatAddOutOfRange
  LDI    -1,R0
_ENDEaseDspFloatOut
  LDI    *-FP(1),R1
  BD     R1
  LDI    *FP,FP
  NOP
  SUBI   2,SP

,   globl _EaseDspFloatIn
,_EaseDspFloatIn
,  PUSH   FP
,  LDI    SP,FP
,
,  LDI    @DUAL,AR0
,  ADDI   *-FP(2),AR0
,  LDF    *AR0,R0
,
,  LDI    *-FP(1),R1
,  BD     R1
,  LDI    *FP,FP
,  NOP
,  SUBI   2,SP

   globl _EaseGetDspPtr
_EaseGetDspPtr
  PUSH   FP
  LDI    SP,FP

  LDI    *-FP(2),R0
  BN     PtrAddOutOfRange
  CMPI   @SIZEOFDUAL,R0
  BGT    PtrAddOutOfRange
  ADDI   @DUAL,R0
```

```
  BR     _ENDEaseGetDspPtr
PtrAddOutOfRange
  LDI    0,R0
_ENDEaseGetDspPtr
  LDI    *-FP(1),R1
  BD     R1
  LDI    *FP,FP
  NOP
  SUBI   2,SP
```

B.2 Application Programming Interface to *Ease*

B.2.1 EaseInit.h

```
/*

FILE   Easeinit h

David Doyle 6/10/94

Date        initials        history
************************************************************
6/10/94      D D             PRE-RELEASE

*/

typedef void (*EaseTaskId_t)(void),

extern void  EaseApplicationTimerInit(int freq),
extern void  EaseSystemTimerInit(int freq),
extern void  EaseCreate(EaseTaskId_t function,
int priority,
int stack_alloc),
```

B.2.2 Ease.h

```
/*

FILE  Ease h


David Doyle 19/8/94

Date        initials        history
************************************************************
19/8/94      D D             PRE-RELEASE

*/

#define TRUE 1
#define FALSE 0
#define NULL (0)
#define NOT_VALID_CHANNEL          -1
#define MSG_TOO_LARGE_FOR_RECEIVER -2
#define INCORRECT_TIMER_MODE       -3
#define RESET_WHILE_ACTIVE         -4

#define INT0_MSG              1
#define END_OF_CONVERSION_INT 2
#define APPLICATION_TIMER_MSG 3
#define SYSTEM_TIMER_MSG      4

#define MONOSTABLE 0
#define ASTABLE 1

typedef void (*EaseTaskId_t)(void),

extern int   EaseReceive(int src_ch,
 void msg[],
 int max_msg_size,
 int *msg_size,
 EaseTaskId_t* rendezvous_tsk),
extern int   EaseSend(int dst_ch,
      void msg[],
      int msg_size,
      EaseTaskId_t* rendezvous_tsk),
extern int   EaseApplicationTimerSet(int ticks,
     int channel,
     int mode),
extern int   EaseSamplerSet(int channel),
extern int   EaseSystemTimerSet(int ticks,
int channel,
int mode_c),
extern void  EaseInt0Init(void),
```

```
extern char* EaseErrorMessage,

extern int EaseClockTick,
extern int EaseInt1Count,
extern int EaseLostInt1,
extern int EaseInt0Count,
extern int EaseLostInt0,
extern int EaseClockTick,
extern int EaseScheduleCount,
extern int EaseSendCount,
extern int EaseReceiveCount,
```

B.2.3 Dsp_if.h

```
/*

  DSP_IF H to be included by both PC and C30 code


    This defines the interface ports and the relevent
  sections of dual port memory for PC interface with C30

    INTEGER reference Values for Dual Port memory used by
  both the C30 and PC  They are relitive to the C30

    Interface routines for C30 to PC

David Doyle

Date        Initials         History
**********************************************************
2/2/94      D D              Pre-Release

*/

#define BASE  0x290
#define DATAL BASE+0
#define DATAH BASE+2
#define ADDRL BASE+4
#define ADDRH BASE+6
#define CTRL  BASE+8
#define INTR  BASE+0xc

#define DUAL         0x00030000
#define INTEGER_OUT  0x200
#define FLOAT_OUT    0x210
#define FLAG_IN      0x222

#define PARM_IN      0x100
#define PARM_OUT     0x130
#define VAR_OUT      0x160

#define ADD_OUT_OF_RANGE  -1

#define EaseError    0x00
#define EaseLock     1
#define EaseUnlock   0

extern int   EaseDspWordOut(int dest,int word),
extern int   EaseDspWordIn(int source),
extern int   EaseDspFloatOut(int dest,float word),
extern void* EaseGetDspPtr(int memref),
```

```
extern int  read_adc(int channel),
extern void outdac(int channel, int value),
```

Bibliography

[1] E W Dijkstra, *Co-operating Sequential Processes" in Programminng Languages* Genuys F (ed) London Academic Press 1965

[2] Wolfgang A Hanlang, Alexander D Stoyenko *Constructing Predictable Real Time Systems* Kluwer Academic Publishers, 1991

[3] David L Ripps *An Implementation Guide to Real-Time Programming* Englewood Cliffs Yourdon Press,1990

[4] Andrew S Tanenbaum *Operating Systems Design and Implementation* Prentice-Hall International Editions, 1987

[5] Ian Pyle, Peter Hurschka, Michel Lissandre and Ken Jackson *Real-Time Systems Investigating Industrial Practice* Wiley Series in Software based Systems, 1993

[6] Andre M van Tilborg *Foundations of Real-Time Computing Formal Specifications and Methods* Kluwer Academic Publishers, 1991

[7] Peter Coad/Edward Yourdon *Object Oriented Analysis second edition* Yourdon Press Computing Series, 1991

[8] Peter Coad/Edward Yourdon *Object Oriented Design* Yourdon Press Computing Series, 1991

[9] Phillip A Laplante *Real-Time Systems Design and Analysis An Engineers Handbook* IEEE Computer Society Press, 1992

[10] Ragunathan Rajkumar *Synchronisation in Real-Time Systems A Priority Inheritance Approach* Kluwer Academic Publishers, 1991

[11] Brian W Kernigan and Dennis M Ritchie *Second Edition The C Programming Language* Prentice Hall Software Series, 1990

[12] Warren Andrews RISC-based boards make headway in real-time applications *Computer Design* (Oct 1991) 69-80

[13] A Steininger and H Schweinzer Can the advantages of RISC be utilized in Real Time Systems? *Proceedings of the Euromicro '91 workshop on Real Time Systems* Paris(1991) 30-35

[14] Texas Instruments *TMS320C3X User's Guide Digital Signal Processing Products 2558539-9721 revision E June 1991* Texas Instruments Incorporated, 1991

[15] Texas Instruments *TMS320C30 Assembly Language Tools User Guide Digital Signal Processing Products* Texas Instruments Incorporated, 1988

[16] Texas Instruments *TMS320C30 Optimising C Compiler Reference Guide Microprocessor Development Systems Products 1604910-9710 revision D August 1990* Texas Instruments Incorporated, 1990

[17] Loughborugh Sound Images Ltd *TMS320C30 PC System Board User Guide & Technical Reference Version 1 01 September 1990* Loughborugh Sound Images Ltd

[18] DIN 44300 Informationsverarbeitung Beuth-Verlag, 1985

[19] G Kalpan The X29 Is it coming or going ? *IEEE Spectrum* pages 54-60, June 1985

[20] G Carlow Architecture of the space shuttle primary avionics software system *Communications of the ACM,* 27(9) 926-936,September 1984

[21] L Motus Time Concepts in Real-Time Software *Control Engineering Practice,*Vol 1, No 1, pp 21-33 1993

[22] B Heindel How to EnsureTime Software Quality in for Real Time Systems *Control Engineering Practice,*Vol 1, No 1, pp 35-41 1993

[23] M Colnaric and W A Halang Architectural Support for Predictability in Hard Real Time Systems *Control Engineering Practice,*Vol 1, No 1, pp 51-57 1993

[24] K Bastiaens and J M Van Campenhout A Visual Real Time Programming Language *Control Engineering Practice,*Vol 1, No 1, pp 59-63 1993

[25] N C Audsley, A Burns and A J Wellings Deadline Monotonic Scheduling Theory and Application *Control Engineering Practice,*Vol 1,No 1,pp 71-78 1993

[26] A Sowyma State Charts-Based Specification and Verification of Real-Time Job Scheduling Systems *Control Engineering Practice,*Vol 1, No 1, pp 107-114 1993

[27] A Mok Fundamental design problems of distributed systems for the hard real-time environment PhD Thesis,MIT Laboratory for Computer science,fsMay 1983

[28] Liu C L and J W Layland Scheduling algorithms for multiprogramming in a hard real-time environment *Jounnal of the ACM 20(1)*1973

[29] E Dijstra The Next Fourty Years *Personal note EWD 1051,* 1989

[30] R Balli, R Curto Dimensioning the active suspension system of a wheeled vehicle *Electrische Bahnen* 1990

[31] L Lewis Optimal Estimation *Wiley* 1986

[32] D Doyle, B Clancy, B Mc Mullin, A Murray Real time Multitasking Executive for Embedded Systems *Proceedings Irish DSP and Control Colloquium* July 1994